Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – hoseong Lee(이호성)

hslee00001@naver.com

a. 소켓 정리

b. 웹서버 만들기

c. 일상생활

소켓

■ sockaddr 구조체

소켓 주소를 표현하는 구조체다.

아래 정의를 보면 주소 체계와 주소 두가지 정보만 갖고 있는 단순한 구조로 되어 있다.

원래 소켓 자체가 TCP/IP만을 목적으로 만들어진 것이 아니어서, 다양한 주소 체계에 맞게 범용 목적으로 사용하기 위해

이런 구조를 가진다.

```
Struct sockaddr{
    sa_famil_t sa_family;     //소켓 주소체계. PF_INET = IPv4 주소 체계
     char sa_data[14];        // 해당 주소체계에서 사용하는 주소 정보
}
```

■ sockaddr_in 구조체

IPv4 주소체계에서 사용하는 구조체다.

소켓 프로그램은 범용 주소 구조체로 sockaddr 을 사용하지만, 주소체계의 종류에 따라 별도의 전용 구조체를 만들어 사용하는 것이 아무래도 편하다. (기타 다른 주소체계 중 local unix 주소체계는 sockaddr_un구조체를 사용함)

소켓 라이브러리는 sockaddr을 사용하므로 라이브러리에 주소 정보를 넘길 때는 sockaddr로 형변환하여 넘긴다.

그러므로 당연히 구조체의 크기는 동일하다.

```
Struct sockaddr_in{
    sin_family_t sin_family;     // IPv4 주소 체계에서 사용하므로 항상 AF_INET로 설정
     unist 16_t sin_port;          // 포트번호
     struct in_addr sin_addr;    // IP주소를 나타내는 32비트 정수 타입구조체
     char siz_zero[8];              // sockaddr 과 같은 크기를 유지하기 위해 필요한 패딩 공간, 항상 0.후농
}
```

■ socklen

_t는 소켓 관련 매개변수에 사용되는 것으로 길이 및 크기 값에 대한 정의를 내려준다. sys/socket.h 헤더파일에 정의 되어있다.

■ pid_t 구조체 는 프로세스번호(PID)를 저장하는 타입(T) 임. 프로세스의 범위가 2~32768 사이의 pid 를 가질 수 있다고 한다.

→ 시스템마다 프로세스번호가 int 형일 수도 있고, 아닐 수 도 있기 때문에 pid_t를 사용

■ sigaction() - signal 함수보다 향상된 기능을 제공하는 시그널 처리를 결정

hole punching

NAT라는 프로토콜이 있으면 이게 사실 아이피를 만든다. 그리고 또다른 나트가 있으면 사설 아이피를 만든다.

이렇게 사설 아이피를 만든 상태에서 두개가 서로 통신을 하려고하면 이시점에서 홀펀칭이 필요하다.

아이피 주소에다가 또다른 아이피 주소로 감싸는것

운영체제

리스소 매니저 - 자원을 효율적으로 관리해준다.

디스크는 소프트웨어로 따지면 파일이다

cpu는 초상화는 프로세스이다.

파일, 프로세스, 페이지

inode

파일을 연다고했을때 디스크에서 읽을 때, 어느위치에 있는지 위치기록이 담겨 있다.

메모리에 올라가야 프로세스이다.

세그먼트 → 가상메모리

task_struct

mm_struct

가상메모리 → 물리메모리 바꾸는 과정이 페이징

tcp-ip분할

init_sock.c

```c
#include "init_sock.h"

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

int init_sock(void)
{
        int sock;

        sock = socket(PF_INET, SOCK_STREAM, 0);

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

        return sock;
}

// serv = 0, clnt = 1
void init_sock_addr(si *serv_addr, int size, char **argv, int opt)
{
        memset(serv_addr, 0, size);
        serv_addr->sin_family = AF_INET;

        if(opt)
        {
                serv_addr->sin_addr.s_addr = inet_addr(argv[1]);
                serv_addr->sin_port = htons(atoi(argv[2]));
        }
        else
        {
                serv_addr->sin_addr.s_addr = htonl(INADDR_ANY);
                serv_addr->sin_port = htons(atoi(argv[1]));
        }
}

void post_sock(int serv_sock, si *serv_addr, int size)
{
        if(bind(serv_sock, (sp)serv_addr, size) == -1)
                err_handler("bind() error!");

        if(listen(serv_sock, 5) == -1)
                err_handler("listen() error!");
}
```

init_sock.h

```c
#ifndef __INIT_SOCK_H__
#define __INIT_SOCK_H__

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

typedef struct sockaddr_in      si;
typedef struct sockaddr *       sp;

void err_handler(char *msg);
int init_sock(void);
void init_sock_addr(si *, int, char **, int);
void post_sock(int, si *, int);

#endif
```

common.h

```c
#ifndef __COMMON_H__
#define __COMMON_H__

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

typedef struct sockaddr_in      si;
typedef struct sockaddr *       sp;

typedef struct __d{
    int data;
    float fdata;
} d;

#define BUF_SIZE                32

#endif
```

basic_serv.c

```c
#include "common.h"
#include "init_sock.h"

int main(int argc, char **argv)
{
        int serv_sock, clnt_sock;
        si serv_addr, clnt_addr;
        socklen_t clnt_addr_size;
        char msg[] = "Hello Network Programming";

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

        serv_sock = init_sock();
        init_sock_addr(&serv_addr, sizeof(serv_addr), argv, 0);
        post_sock(serv_sock, &serv_addr, sizeof(serv_addr));

        clnt_addr_size = sizeof(clnt_addr);
        clnt_sock = accept(serv_sock, (sp)&clnt_addr, &clnt_addr_size);

        if(clnt_sock == -1)
                err_handler("accept() error");

        write(clnt_sock, msg, sizeof(msg));

        close(clnt_sock);
        close(serv_sock);

        return 0;
}
```

basic_clnt.c

```c
#include "common.h"
#include "init_sock.h"

int main(int argc, char **argv)
{
        int sock, len;
        si serv_addr;
        char msg[32] = {0};

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

        sock = init_sock();
        init_sock_addr(&serv_addr, sizeof(serv_addr), argv, 1);

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

        len = read(sock, msg, sizeof(msg) - 1);

        if(len == -1)
                err_handler("read() error!");

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

        return 0;
}
```

함수를 분리해놓는 목적은 재활용이다.

웹서버 만들기

Webserv.c

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

#define BUF_SIZE      1024
#define SMALL_BUF     100

typedef struct sockaddr_in    si;
typedef struct sockaddr *     sp;

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

void send_error(FILE *fp)
{
    char protocol[]= "HTTP/1.0 400 Bad Request\r\n";
    char server[] = "Server:Linux Web Server\r\n";
    char cnt_len[] = "Content-length:2048\r\n";
    char cnt_type[] = "Content-type:text/html\r\n\r\n";
    char content[] = "<html><head><title>Network</title></head>"
    "<body><font size=+5><br> 오류 발생! 요청 파일명 및 방식 확인!""</font></body></html>";

    fputs(protocol, fp);
    fputs(server, fp);
    fputs(cnt_len, fp);
    fputs(cnt_type, fp);
    fflush(fp);
}
char *content_type(char *file)
{
    char extension[SMALL_BUF];
    char file_name[SMALL_BUF];
    strcpy(file_name, file);
    strtok(file_name, ".");
    strcpy(extension,strtok(NULL,"."));

    if(!strcmp(extension,"html") || !strcmp(extension, "htm"))
        return "text/html";
    else
        return "text/plain";
}
```

```c
void send_data(FILE *fp, char *ct, char *file_name)
{
    char protocol[] = "HTTP/1.0 200 OK\r\n";
    char server[] = "Server:Linux Web Server\r\n";
    char cnt_len[] = "Content-length:2048\r\n";
    char cnt_type[SMALL_BUF];
    char buf[BUF_SIZE];
    FILE *send_file;

    sprintf(cnt_type, "Content-type:%s\r\n\r\n",ct);
    send_file = fopen(file_name,"r");

    if(send_file == NULL)
    {
        send_error(fp);
        return;
    }
    fputs(protocol,fp);
    fputs(server, fp);
    fputs(cnt_len, fp);
    fputs(cnt_type, fp);

    while(fgets(buf, BUF_SIZE, send_file) != NULL)
    {
        fputs(buf,fp);
        fflush(fp);
    }
    fflush(fp);
    fclose(fp);
}
```

```c
void *request_handler(void *arg)
{
    int clnt_sock = *((int *)arg);
    char req_line[SMALL_BUF];
    FILE *clnt_read;
    FILE *clnt_write;

    char method[10];
    char ct[15];
    char file_name[30];

    clnt_read = fdopen(clnt_sock,"r");
    clnt_write = fdopen(dup(clnt_sock),"w");
    fgets(req_line, SMALL_BUF, clnt_read);

    if(strstr(req_line, "HTTP/") == NULL)
    {
        send_error(clnt_write);
        fclose(clnt_read);
        fclose(clnt_write);
        return NULL;
    }

    strcpy(method, strtok(req_line, " /"));
    strcpy(file_name, strtok(NULL," /"));
    strcpy(ct, content_type(file_name));

    if(strcmp(method, "GET") != 0)
    {
        send_error(clnt_write);
        fclose(clnt_read);
        fclose(clnt_write);
        return NULL;
    }
    fclose(clnt_read);
    send_data(clnt_write, ct, file_name);
}
```

```c
int main(int argc, char **argv)
{
    int serv_sock, clnt_sock;
    si serv_addr, clnt_addr;
    int clnt_addr_size;
    char buf[BUF_SIZE];
    pthread_t t_id;

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

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    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)
        error_handling("bind() error");
    if(listen(serv_sock, 20) == -1)
        error_handling("listen() error");

    for(;;)
    {
        clnt_addr_size = sizeof(clnt_addr);
        clnt_sock = accept(serv_sock,(sp)&clnt_addr, &clnt_addr_size);
        printf("Connection Request: %s:%d\n",inet_ntoa(clnt_addr.sin_addr)
           , ntohs(clnt_addr.sin_port));
        pthread_create(&t_id, NULL, request_handler, &clnt_sock);
        pthread_detach(t_id);
    }
    close(serv_sock);
    return 0;
}
```

network.h

```c
#ifndef __NETWORK_H__
#define __NETWORK_H__

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

typedef struct sockaddr_in si;
typedef struct sockaddr *       sp;
typedef struct timeval tv;

typedef struct __data{
        int clnt_sock;
        si n_clnt_addr;

}net_data;
void err_handler(char *);
double get_runtime(tv,tv);
void send_msg(char*, int , int*);
void *clnt_handler(void*);
void block_ip_init(void);
void block_ip_check(int, si);

#define BUF_SIZE        10000
#define MAX_CLNT    10000
#define NAME_SIZE   32
#endif
```

first.html

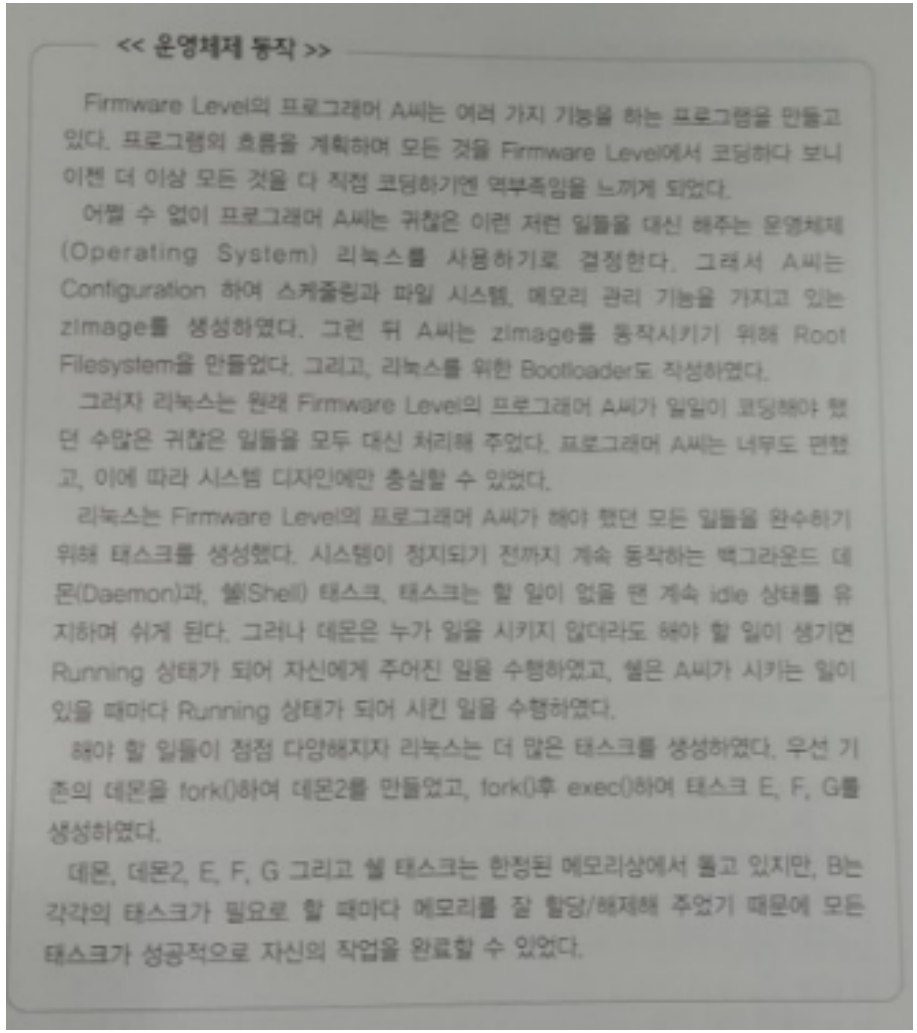
```html
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>성환이형 컴퓨터좀 바꾸세요</h1>
<p>lg  저 같으면 갖다 버립니다.</p>

</body>
</html>
```

실행

./webserv 6666

브라우저 : 127.0.0.1:6666/first.html

chapter 0



<< 운영체제 동작 >>

Firmware Level의 프로그래머 A씨는 여러 가지 기능을 하는 프로그램을 만들고 있다. 프로그램의 흐름을 계획하여 모든 것을 Firmware Level에서 코딩하다 보니 이젠 더 이상 모든 것을 다 직접 코딩하기엔 역부족임을 느끼게 되었다.

어쩔 수 없이 프로그래머 A씨는 귀찮은 이런 저런 일들을 대신 해주는 운영체제(Operating System) 리눅스를 사용하기로 결정한다. 그래서 A씨는 Configuration 하여 스케줄링과 파일 시스템, 메모리 관리 기능을 가지고 있는 zImage를 생성하였다. 그런 뒤 A씨는 zImage를 동작시키기 위해 Root Filesystem을 만들었다. 그리고, 리눅스를 위한 Bootloader도 작성하였다.

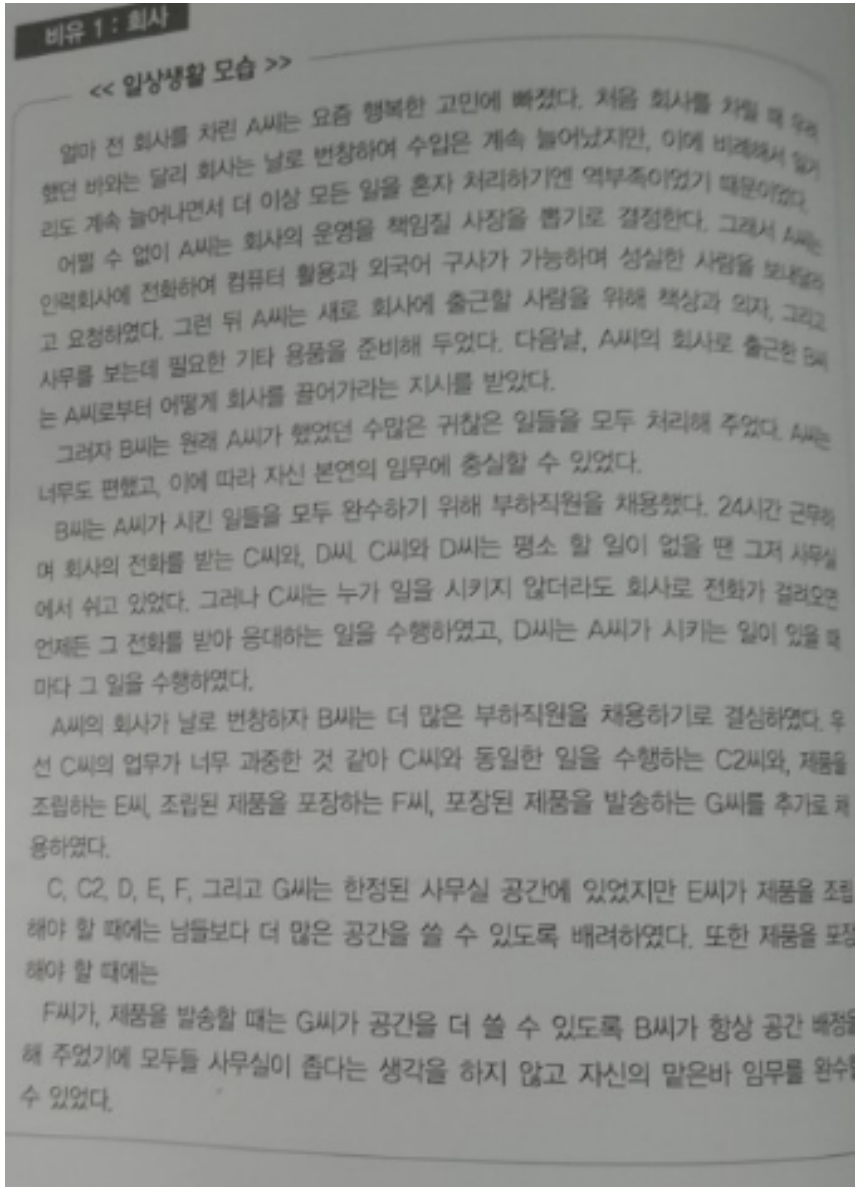
그러자 리눅스는 원래 Firmware Level의 프로그래머 A씨가 일일이 코딩해야 했던 수많은 귀찮은 일들을 모두 대신 처리해 주었다. 프로그래머 A씨는 너무도 편했고, 이에 따라 시스템 디자인에만 충실할 수 있었다.

리눅스는 Firmware Level의 프로그래머 A씨가 해야 했던 모든 일들을 완수하기 위해 태스크를 생성했다. 시스템이 정지되기 전까지 계속 동작하는 백그라운드 데몬(Daemon)과, 쉘(Shell) 태스크. 태스크는 할 일이 없을 땐 계속 idle 상태를 유지하며 쉬게 된다. 그러나 데몬은 누가 일을 시키지 않더라도 해야 할 일이 생기면 Running 상태가 되어 자신에게 주어진 일을 수행하였고, 쉘은 A씨가 시키는 일이 있을 때마다 Running 상태가 되어 시킨 일을 수행하였다.

해야 할 일들이 점점 다양해지자 리눅스는 더 많은 태스크를 생성하였다. 우선 기존의 데몬을 fork()하여 데몬2를 만들었고, fork()후 exec()하여 태스크 E, F, G를 생성하였다.

데몬, 데몬2, E, F, G 그리고 쉘 태스크는 한정된 메모리상에서 돌고 있지만, B는 각각의 태스크가 필요로 할 때마다 메모리를 잘 할당/해제해 주었기 때문에 모든 태스크가 성공적으로 자신의 작업을 완료할 수 있었다.

비유 1: 회사



비유 1 : 회사

<< 일상생활 모습 >>

얼마 전 회사를 차린 A씨는 요즘 행복한 고민에 빠졌다. 처음 회사를 차릴 때 우려했던 바와는 달리 회사는 날로 번창하여 수입은 계속 늘어났지만, 이에 비례해서 일거리도 계속 늘어나면서 더 이상 모든 일을 혼자 처리하기엔 역부족이었기 때문이었다.

어쩔 수 없이 A씨는 회사의 운영을 책임질 사장을 뽑기로 결정한다. 그래서 A씨는 인력회사에 전화하여 컴퓨터 활용과 외국어 구사가 가능하며 성실한 사람을 보내줄 것을 고 요청하였다. 그런 뒤 A씨는 새로 회사에 출근할 사람을 위해 책상과 의자, 그리고 사무를 보는데 필요한 기타 용품을 준비해 두었다. 다음날, A씨의 회사로 출근한 B씨는 A씨로부터 어떻게 회사를 끌어가라는 지시를 받았다.

그러자 B씨는 원래 A씨가 했었던 수많은 귀찮은 일들을 모두 처리해 주었다. A씨는 너무도 편했고, 이에 따라 자신 본연의 임무에 충실할 수 있었다.

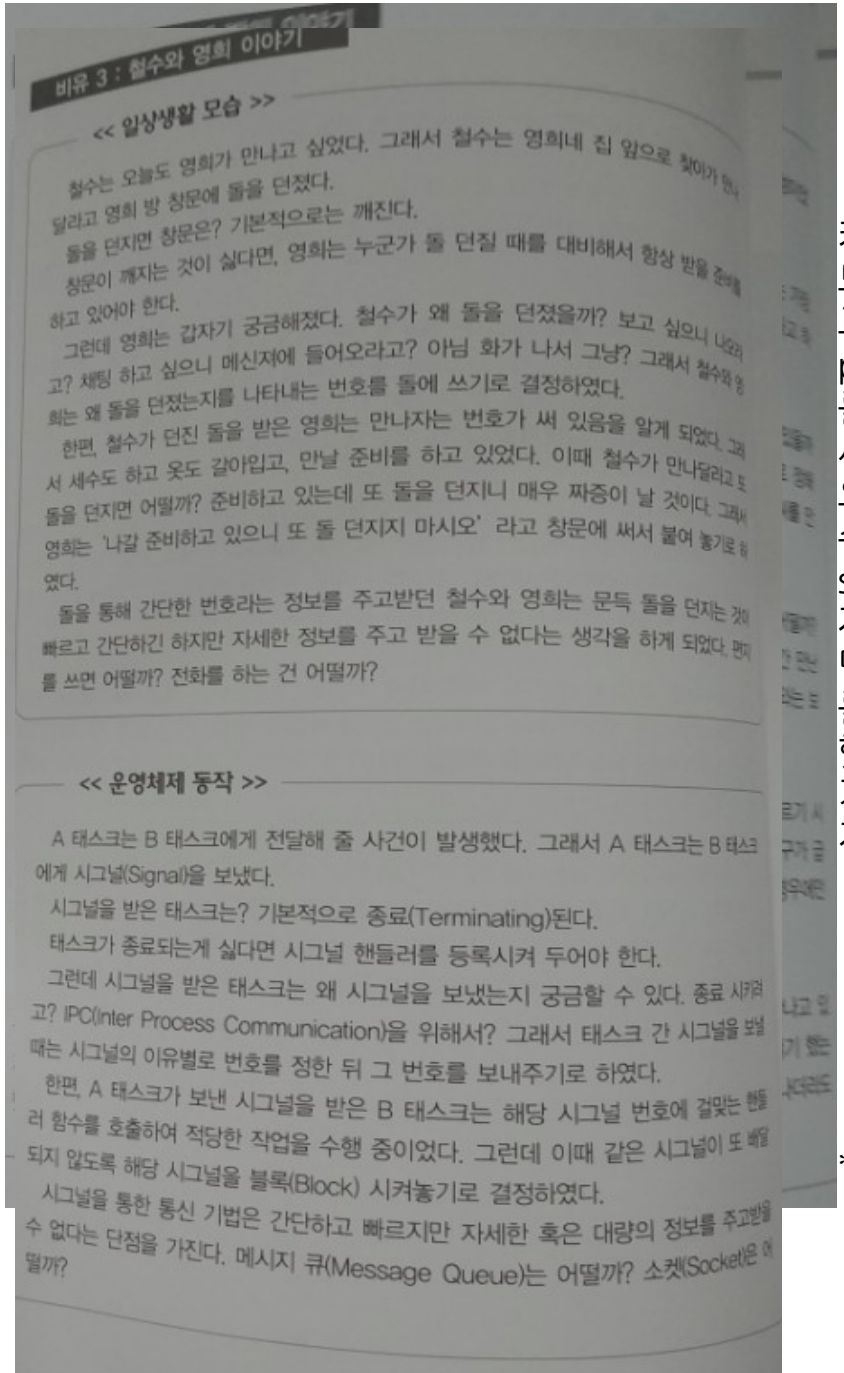
B씨는 A씨가 시킨 일들을 모두 완수하기 위해 부하직원을 채용했다. 24시간 근무하며 회사의 전화를 받는 C씨와, D씨. C씨와 D씨는 평소 할 일이 없을 땐 그저 사무실에서 쉬고 있었다. 그러나 C씨는 누가 일을 시키지 않더라도 회사로 전화가 걸려오면 언제든 그 전화를 받아 응대하는 일을 수행하였고, D씨는 A씨가 시키는 일이 있을 때마다 그 일을 수행하였다.

A씨의 회사가 날로 번창하자 B씨는 더 많은 부하직원을 채용하기로 결심하였다. 우선 C씨의 업무가 너무 과중한 것 같아 C씨와 동일한 일을 수행하는 C2씨와, 제품을 조립하는 E씨, 조립된 제품을 포장하는 F씨, 포장된 제품을 발송하는 G씨를 추가로 채용하였다.

C, C2, D, E, F, 그리고 G씨는 한정된 사무실 공간에 있었지만 E씨가 제품을 조립해야 할 때에는 남들보다 더 많은 공간을 쓸 수 있도록 배려하였다. 또한 제품을 포장해야 할 때에는

F씨가, 제품을 발송할 때는 G씨가 공간을 더 쓸 수 있도록 B씨가 항상 공간 배정을 해 주었기에 모두들 사무실이 좁다는 생각을 하지 않고 자신의 맡은바 임무를 완수할 수 있었다.

비유 2: 카사노바 박씨 이야기



비유 3 : 철수와 영희 이야기

<< 일상생활 모습 >>

철수는 오늘도 영희가 만나고 싶었다. 그래서 철수는 영희네 집 앞으로 찾아가 전화
달라고 영희 방 창문에 돌을 던졌다.
돌을 던지면 창문은? 기본적으로는 깨진다.
창문이 깨지는 것이 싫다면, 영희는 누군가 돌 던질 때를 대비해서 항상 받을 준비를
하고 있어야 한다.
그런데 영희는 갑자기 궁금해졌다. 철수가 왜 돌을 던졌을까? 보고 싶으니 나오라
고? 채팅 하고 싶으니 메신저에 들어오라고? 아님 화가 나서 그냥? 그래서 철수와 영
희는 왜 돌을 던졌는지를 나타내는 번호를 돌에 쓰기로 결정하였다.
한편, 철수가 던진 돌을 받은 영희는 만나자는 번호가 써 있음을 알게 되었다. 그래
서 세수도 하고 옷도 갈아입고, 만날 준비를 하고 있었다. 이때 철수가 만나달라고 또
돌을 던지면 어떨까? 준비하고 있는데 또 돌을 던지니 매우 짜증이 날 것이다. 그래서
영희는 '나갈 준비하고 있으니 또 돌 던지지 마시오' 라고 창문에 써서 붙여 놓기로 하
였다.
돌을 통해 간단한 번호라는 정보를 주고받던 철수와 영희는 문득 돌을 던지는 것이
빠르고 간단하긴 하지만 자세한 정보를 주고 받을 수 없다는 생각을 하게 되었다. 편지
를 쓰면 어떨까? 전화를 하는 건 어떨까?

<< 운영체제 동작 >>

A 태스크는 B 태스크에게 전달해 줄 사건이 발생했다. 그래서 A 태스크는 B 태스크
에게 시그널(Signal)을 보냈다.
시그널을 받은 태스크는? 기본적으로 종료(Terminating)된다.
태스크가 종료되는게 싫다면 시그널 핸들러를 등록시켜 두어야 한다.
그런데 시그널을 받은 태스크는 왜 시그널을 보냈는지 궁금할 수 있다. 종료 시키려
고? IPC(Inter Process Communication)을 위해서? 그래서 태스크 간 시그널을 보낼
때는 시그널의 이유별로 번호를 정한 뒤 그 번호를 보내주기로 하였다.
한편, A 태스크가 보낸 시그널을 받은 B 태스크는 해당 시그널 번호에 걸맞는 핸들
러 함수를 호출하여 적당한 작업을 수행 중이었다. 그런데 이때 같은 시그널이 또 배달
되지 않도록 해당 시그널을 블록(Block) 시켜놓기로 결정하였다.
시그널을 통한 통신 기법은 간단하고 빠르지만 자세한 혹은 대량의 정보를 주고받을
수 없다는 단점을 가진다. 메시지 큐(Message Queue)는 어떨까? 소켓(Socket)은 어
떨까?

카사노바 램은 사실 카사노바가 아니다. 전설의 카사노바는 램의 친구 하드디스크이다. 하드디스크는 친구 램에게 자신의 여자친구를 빌려주겠다고 하고, 고르라고 얘기했다. (둘다쓰레기다.) 램은 24명의 어떤 process 고를지 생각 중이다. 친구인 cpu은 그래픽이 죽이는 프로세스를 선택하라 했다. 주인 격인 하드디스크는 지금 하고싶은(데이트) 프로세스를 고르라고 얘기했다. 램은 24개의 프로세스 중 가장 자신에게 중요하다고 생각되는 우선순위를 정하고 모두 만나기로 생각했다. 그리고 수행할 작업이 많고 적음으로 만나는 시간도 정했다. (context switching) 이 때, 다른여자친구와 놀고 있을 때 (run queue), 다른여자친구들 중(wait queue) 길거리에서 우연히 다른여자친구의 아버지를 만났거나, 다른 여자친구가 사고를 당했을 때, 그 프로세스를 인터럽트를 주어 당장 실행시켜줬다.(system call). 불안요소가 많았던 램은 생각했다. 램의 뇌 리눅스는 한 프로세스를 만났을 때, 그녀에게 최선을 다하기로 마음을 먹었다. Context switching 을 할 때, 데이터가 서로 꼬이지 않기 위해 lock을 걸어서 임계 영역을 보호했다.

** IPC (Inter-Process Communication) 란??

프로세스 사이의 데이터통신은 IPC를 통해서 한다.
커널은 프로세스 통신을 위해 다양한 IPC매커니즘을 제공한다.
1. 시그널 (Signal)
2. 파이프 (Pipe)
3. 메세지 큐 (Message Queue)
4. 공유 메모리 (Shared Memory)
5. 메일박스 (Mailbox)

1. 시그널
시그널은 임의의 프로세스에세 특정 이벤트가 발생했다는 사실을 알려주는 매커니즘
(Ctrl + C 가 눌리면 SIGINT시그널이 프로세스에 전달되어 중단되는 것)

시그널 전달이 이뤄지는 과정
PCB(Process Control Block ; 리눅스에서 task_struct) 에 시그널 개수만큼 미리 확보된 시그널 벡터가 있다. 시그널 개수만큼 확보되어있기 때문에, 평소에는 0으로 clear되어 있다가 해당 시그널이 발생하면 그 bit만 1로 set 해주면 되는것(즉, 3번 시그널이 전달되면 3번 시그널벡터가 1로 set) 시그널이 발생하면, 프로세스는 기존의 수행 흐름에서 벗어나, 해당 시그널 핸들러를 수행한다.

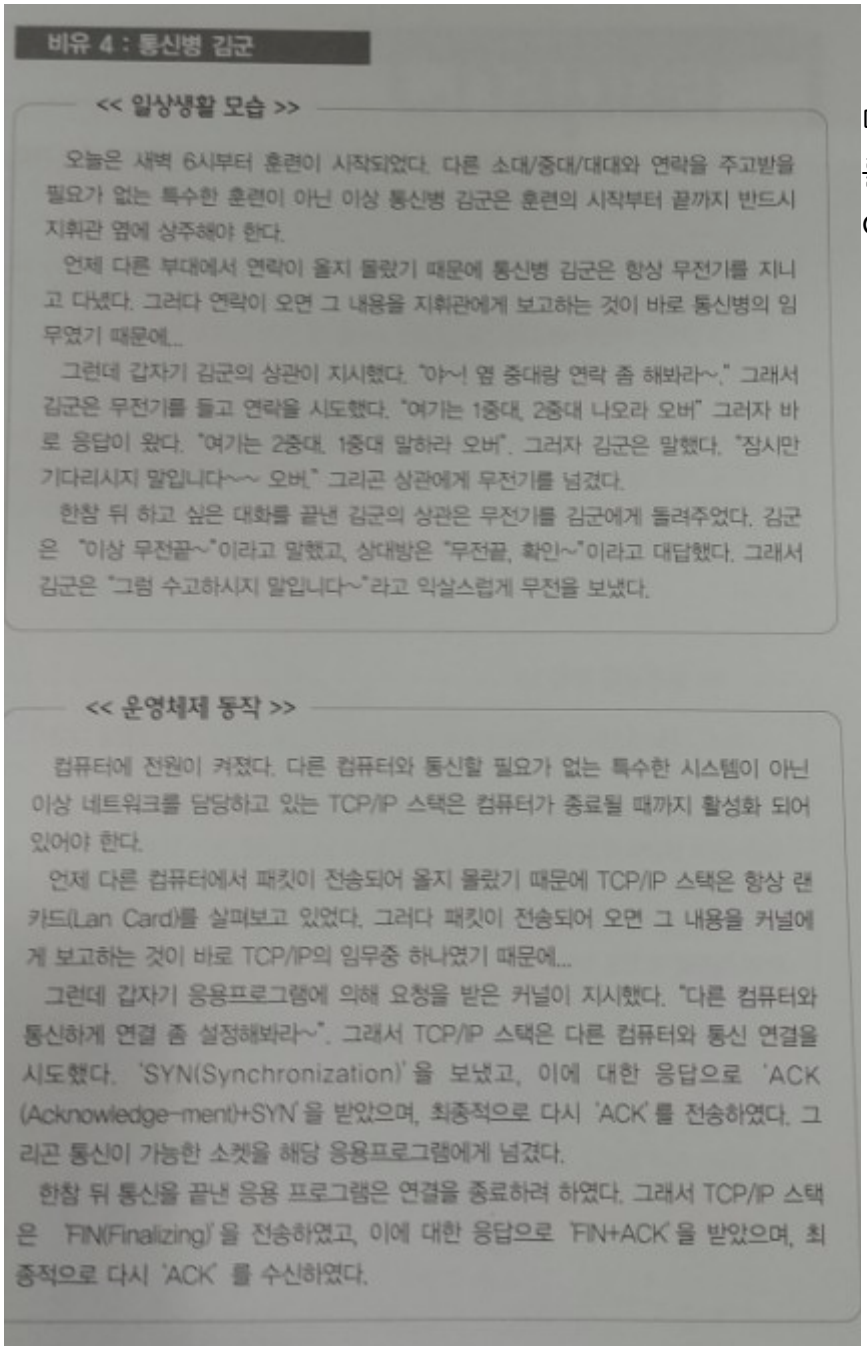
시그널 핸들러는 커널이 지정해놓을 수도 있고(예를들어 프로세스 종료, 시그널 무시, 프로세스 중지, 실행 재개 등등) 사용자가 특정 기능을 수행하는 코드도 등록해 놓을 수 있다.

(시그널 핸들러는 시그널 공간 할당 후에 함수포인터를 만들어서 시그널을 처리할 함수의 주소를 가리키도록 해놓는 방식으로 구현하게 된다.)

2. 파이프

파이프는 프로세스 사이에 형식없는(unstructured) 데이터의 교환을 가능케 합니다. 파이프를 통해 전달되는 데이터는 단순한 바이트 스트림 형태로 전달됩니다.
파이프는 말그대로 파이프처럼 생겼다고 생각하시면 됩니다.
프로세스1 => 파이프 => 프로세스2 요렇게 된다.
리눅스에서 쉘명령어중 | <- 요것도 파이프입니다.
많이들 쓰시다시피 ls | grep ~~ 이런식으로 많이 쓴다. 이것의 뜻은 "ls의 출력을 grep의 입력으로!" 라는 뜻이다.
파이프의 특징은 다음과 같다.
- 파이프 안의 데이터는 바이트스트림(형식 없는 데이터)로 취급되기 때문에, 여러 메세지 객체가 쓰여졌을 때, 메세지의 경계를 분간할 수 없다.
- 또한 파이프는 FIFO(First In First Out)방식으로 데이터를 처리하므로, 메세지가 파이프에 들어온 순서대로 처리됩니다.
- 파이프는 고정된 크기(용량)을 갖는다.

- 파이프의 데이터가 용량을 초과하면, 파이프에 데이터를 쓰는 프로세스는 "블록" 상태가 됩니다. 파이프에 쓰려고 하는데 파이프 용량이 꽉 찼으니 파이프가 데이터 쓰기 가능 상태가 되기 전까지 블록된다.
- 반대로, 파이프가 텅 비어있으면, 파이프에서 데이터를 읽는 프로세스는 "블록" 상태가 됩니다. 파이프에서 데이터 읽으려고 하는데 파이프가 비어있으니, 파이프에 데이타가 쓰여져서 그걸 읽기전까지 블록된다.



비유 4 : 통신병 김군

<< 일상생활 모습 >>

오늘은 새벽 6시부터 훈련이 시작되었다. 다른 소대/중대/대대와 연락을 주고받을 필요가 없는 특수한 훈련이 아닌 이상 통신병 김군은 훈련의 시작부터 끝까지 반드시 지휘관 옆에 상주해야 한다.

언제 다른 부대에서 연락이 올지 몰랐기 때문에 통신병 김군은 항상 무전기를 지니고 다녔다. 그러다 연락이 오면 그 내용을 지휘관에게 보고하는 것이 바로 통신병의 임무였기 때문에...

그런데 갑자기 김군의 상관이 지시했다. "야~! 옆 중대랑 연락 좀 해봐라~." 그래서 김군은 무전기를 들고 연락을 시도했다. "여기는 1중대, 2중대 나오라 오버" 그러자 바로 응답이 왔다. "여기는 2중대, 1중대 말하라 오버". 그러자 김군은 말했다. "잠시만 기다리시지 말입니다~~ 오버." 그리곤 상관에게 무전기를 넘겼다.

한참 뒤 하고 싶은 대화를 끝낸 김군의 상관은 무전기를 김군에게 돌려주었다. 김군은 "이상 무전끝~"이라고 말했고, 상대방은 "무전끝, 확인~"이라고 대답했다. 그래서 김군은 "그럼 수고하시지 말입니다~"라고 익살스럽게 무전을 보냈다.

<< 운영체제 동작 >>

컴퓨터에 전원이 켜졌다. 다른 컴퓨터와 통신할 필요가 없는 특수한 시스템이 아닌 이상 네트워크를 담당하고 있는 TCP/IP 스택은 컴퓨터가 종료될 때까지 활성화 되어 있어야 한다.

언제 다른 컴퓨터에서 패킷이 전송되어 올지 몰랐기 때문에 TCP/IP 스택은 항상 랜카드(Lan Card)를 살펴보고 있었다. 그러다 패킷이 전송되어 오면 그 내용을 커널에게 보고하는 것이 바로 TCP/IP의 임무중 하나였기 때문에...

그런데 갑자기 응용프로그램에 의해 요청을 받은 커널이 지시했다. "다른 컴퓨터와 통신하게 연결 좀 설정해봐라~". 그래서 TCP/IP 스택은 다른 컴퓨터와 통신 연결을 시도했다. 'SYN(Synchronization)'을 보냈고, 이에 대한 응답으로 'ACK(Acknowledge-ment)+SYN'을 받았으며, 최종적으로 다시 'ACK'를 전송하였다. 그리곤 통신이 가능한 소켓을 해당 응용프로그램에게 넘겼다.

한참 뒤 통신을 끝낸 응용 프로그램은 연결을 종료하려 하였다. 그래서 TCP/IP 스택은 'FIN(Finalizing)'을 전송하였고, 이에 대한 응답으로 'FIN+ACK'을 받았으며, 최종적으로 다시 'ACK'를 수신하였다.

Server 에서 리슨을 하여 클라이언트로 붙터 정보가 전달되는지 계속 기다린다. 클라이언트에서 요청이오면 accept으로 read를 하고, write 로 메세지를 클라이언트에게 전달한다. 그다음 클라이언트 측에서도 리드를 하고 서로 close하여 연결을 끊는다.

비유 5 : 크게 성공할 미래의 주방장 이군

<< 일상생활 모습 >>

식사 시간이 되어 주방장은 주방 보조 이군을 불렀다. "야~! 어서 창고에 가서 재료 가져와" 라고 이군에게 지시했다.

이군은 급히 창고로 이동하여 식량을 찾기 시작했다. 사실 이군은 전혀 당황하지 않았다. 각종 물건들을 창고 안에 여기 저기 대충 쌓아 놓으면 나중에 찾기가 불편할 것이라고 판단한 이군은 평소 물건을 창고에 넣어둘 때 기가 막히게 정리를 해두었던 것이다.

우선 창고의 문 앞에는 창고 안에 어떤 물건들이 들어있고, 어떤 방식으로 정리되어 있으며, 또한 창고 안 어디에 빈 공간이 있는지를 나타내는 장부를 붙여 두었고, 창고 내부는 저장한 물건 종류별로 정리가 되어 있었으며, 각 종류별로 어떤 물건이 얼마나 있는지를 나타내는 정리표를 완벽하게 만들어서 붙여 놓았다.

<< 운영체제 동작 >>

커널은 파일시스템(File System)에게 A.txt라는 파일을 읽어오라고 명령을 내렸다.

파일시스템은 급히 하드디스크에서 A.txt 라는 파일을 찾기 시작했다. 사실 파일시스템은 전혀 당황하지 않았다. 각종 파일들을 하드디스크 안에 여기 저기 대충 저장해 놓으면 나중에 찾기가 불편할 것이라고 판단한 파일시스템은 평소 파일을 디스크에 저장할 때 Ext2(혹은 FAT, 혹은 LFS, 혹은 여러분이 알고 있는 어떠한 파일시스템) 형식에 맞게 저장해 두었던 것이다.

우선 하드디스크의 제일 첫 부분에는 수퍼블록(Super Block)을 통해 해당 하드디스크를 위해 구축된 파일시스템의 전역적인 메타데이터를 담았고, 블록 할당/해제 정보를 담아 놓는 영역도 기록해 두었으며, 나머지 하드디스크 공간은 디렉토리 개념을 통해 정리가 되어 있었고, 파일 별로 메타데이터를 기록해 두었다.