TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-04-05 (31 회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: **정유경**
ucong@naver.com

1. Basic server, Basic client 를 파일분할써서 기능단위로 분리하기

/*basic_client*/

```
#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;
}
```

/* basic_serv*/

```
#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;
}
```

/* init_sock.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);
```

/* init_sock.h */

```
#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
```

```
        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!");
}
```

2. C 로 웹 서버 구현하기 - web_serv.c web_clnt.c , first.html

First.html

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

<h1>Heading:Look at here!</h1>
<p> paragraph: Hello! My first HTML</p>

</body>
</html>
~
```

```
/*web_serv.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_handler(char *msg)
{
        fputs(msg,stderr);
        fputc('₩n', stderr);
        exit(1);
```

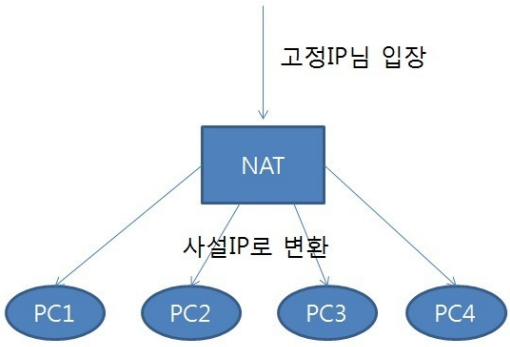
```
}

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〉 오류발생! 요청 파일명
및 방식확인!" "〈/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";
}

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);
}

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);
}

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_handler("bind() error");
	if(listen(serv_sock, 20)==-1)
		error_handler("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;
}
```

3. UDP/ TCP 의 Hole Punching(홀 펀칭) 기법 & NAT 프로토콜 및 터미널링

현재 인터넷 주소체계의 방식은 global 주소영역, private 주소영역들이 네트워크주소변환(NAT)를통하여 연결되는 주소체계이다. 서로 다른 private 네트워크에 존재하는 두 peer 가 직접적으로통신하게 하는 방법이 필요하다. 네트워크에서 구멍을 뚫는다는 의미는 NAT 라는 네트워크 장비를 두고 있어서 직접적인 TCP/UDP 통신이 불가능한 Peer 들간에 직접적인 통신이 가능할 수 있도록 (IP, PORT) 확보하는 것을 말함



홀펀칭이 필요한 이유는 P2P (Peer-to-Peer) 기반의 게임이나 통신 방식에서는 클라이언트끼리 직접 연결을 맺고 통신을 진행하여야 하는데, 참여하는 클라이언트가 NAT 하위에 존재한다면 직접 연결이 불가능하기 때문.

(사설 IP 로 서로 통신을 하게 되면 같은 NAT 안에서의 통신은 가능 하지만 서로 다른 NAT 에서는 통신이 되지 않음. NAT 안에서만 존재하는 가상 IP 이기 때문. NAT 를 거치게 되면 사설 IP(192.168.0.2 이런식의 IP 가 지급)

홀펀칭 과정

1. 클라이언트는 gethostname 으로 사설 or 공인 IP 의 주소를 저장함.
2. NAT 환경이 아닌 서버에게 dummy Packet 을 보냄.
3. 서버는 받은 패킷의 IP, PORT 로 IP, PORT 를 패킷 내용에 담아 클라이언트에게 보냄.
4. 클라이언트는 사설 IP, 공인 IP 모두 저장.
5. 클라이언트는 수집된 사설 IP, 공인 IP 를 서버에게 보냄.
6. 파티가 이루어질 경우 서버는 각각의 Peer 에게 상대방의 사설,공인 IP 를 전송.
7. 각각의 Peer 들은 일정시간동안 사설 IP 로 패킷 전송.
8. 사설 IP 로 패킷 전송 실패시 공인 IP 로 패킷 전송.
9. 공인 IP 로 패킷 전송 실패시 최후의 보루 릴레이서버에게 맡김.

4. [과제] 리눅스 커널 내부구조 (교재 16~22p)

비유 1

리눅스는 커널 내부에서 태스크라는 객체로 프로세스나 쓰레드를 관리. (각 프로세스 혹은 쓰레드마다 task_struct 자료구조를 생성해 관리) 커널은 각 태스크를 task_struct 내의 pid 필드로 구분. 여기서 task_struct 에는 쓰레드를 그룹으로 관리하는 tgid : Thread Group ID, 라는 필드가 따로 존재. (pid 필드와 tgid 필드는 별개) 이때 한 프로세스 안의 여러 개의 쓰레드들은 동일한 pid 를 공유. 커널은 프로세스가 생성될 때는 유일한 pid 를 할당하고 (이때 프로세스에 할당된 pid 와 tgid 는 같은 값.) 커널이 쓰레드를 생성할 때의 그 쓰레드의 tgid 값은 부모 쓰레드의 tgid 값과 동일한 값으로 생성.

task_struct 자료구조와 태스크의 가상 주소 공간의 관계를 보면, 가상 메모리 관련 정보는 task_struct 안의 mm 필드에서 관리 프로그램이 시작되면 태스크가 생성되고 그 태스크에게 가상 주소 공간을 제공. 이때 물리 메모리의 일부를 할당하여 태스크가 원하는 디스크 상의 내용을 물리메모리에 적재. 이렇게 물리 메모리의 실제 주소와 태스크의 가상 주소 공간이 연결됨 (물리메모리의 page frame 과 마찬가지로 가상메모리도 일반적으로 4KB 의 페이지 단위의 고정된 크기로 할당) 가상 메모리 기법의 도입으로 필요한 메모리만 물리 메모리에 적재되므로, 수행에 필요한 일부 페이지만을 적재하고 나머지는 필요할 때 demand paging 으로 물리 메모리의 필요한 부분만을 사용하여 많은 태스크들이 동시에 물리메모리에 적재될 수 있게 됨

비유 2

리눅스 운영체제는 프로세스나 스레드에 대해 선점(Preemption) 스케줄링 기법을 제공. 선점 스케줄링은 시분할 시스템에서 한 프로세스의 CPU 독점을 방지하기 위해 주어지는 타임 슬라이스(Time slice)가 소진되었거나, 인터럽트나 시스템 호출 종료 시에 더 높은 우선 순위의 프로세스가 발생하였음을 알았을 때 현 실행 프로세스로부터 강제로 CPU 를 회수하여 다른 프로세스에 할당하는 것.

프로세스는 실행되는 동안 커널이 가진 자원(CPU, Memory, Devices, Files)을 차지하기 위해 경쟁하고 커널은 이들에게 효율적으로 자원을 스케줄링하여 할당하고 회수. 프로세스는 커널이 가진 여러가지 자원의 할당 및 사용을 위해 커널 함수를 호출해야 하는데, 이러한 커널 함수들을 일반적으로 시스템 호출(System call)이라 함.

프로세스나 스레드는 리눅스 커널 내부에서 모두 자원을 차지하기 위해 서로 경쟁하는 태스크들로 관리
프로세스의 상태는 다음의 3 가지 → 실행(Running) 상태: 프로세스에 CPU 가 할당되어 실행 중인 상태, 준비(Ready) 상태: 커널에 의해 스케줄링되어 CPU 가 할당되면 실행될 수 있는 상태. 대기(Blocked) 상태: 커널은 한 프로세스의 대기 상태 동안 CPU 를 다른 프로세스에 할당하여 효율성을 높임

CPU 할당을 기다리는 프로세스들은 스케줄링 큐(Scheduling Queue) → 런큐에서 대기하다가 스케줄러에 의해 할당받게 됨. 커널은 CPU 를 차지하여 실행 중인 프로세스의 CPU 독점을 방지하기 위해 Time slice 에 의한 시분할 시스템의 개념을 도입하는 것이 보통. Time slice 란 프로세스의 CPU 독점을 방지하기 위해 매 CPU 차지시마다 CPU 사용의 한계 구간으로 주어지는 것으로 커널은 타임 슬라이스를 다 사용한 프로세스에서 일단 CPU 를 회수하고, 다른 프로세스의 CPU 사용이 차례로 이루어진 후에 다시 CPU 를 할당함.

리눅스의 경우 프로세스에 관한 커널의 모든 정보는 태스크 구조체(task_struct)에 저장, 실행상태와 준비상태의 프로세스의 태스크 구조체들은 모두 스케줄링 큐에 연결 리스트 형태로 저장. 프로세스 상태가 대기 중일때, 그 태스크 구조체는 스케줄링 큐에서 제거되어 특정 대기큐에 소속. 입출력이 완료 되거나 기다리던 이벤트가 발생하면 대기 상태의 프로세스는 다시 준비(Ready) 상태가 되어 그 태스크 구조체를 스케줄링 큐로 복귀. 프로세스의 수행이 종료되면 모든 포르세스의 task 구조체와 메모리 영역의 프로그램들은 제거되어야 함. 그러나 리눅스 커널에서는 해당 프로세스를 생성한 후, 생성 프로세스의 종료를 기다리는 부모 프로세스에게 종료 프로세스에 대한 정보를 전달해야 하므로, 정보 전달 시까지 task 구조체를 유지. 이렇게 종료는 되었지만 task 구조체는 유지하고 있는 상태를 좀비 상태라고 함.

프로세스는 수행 중에 자원에 대한 할당 대기와 외부 인터럽트 처리와 같은 작업에 의해 언제든지 수행이 중지되고, 그 후에 다시 속개되는 일이 반복 따라서 프로세스의 중지 시점에는 프로세스의 실행에 필요한 모든 정보와 환경들이 저장되어야 하고, 속개시에는 중단 시점의 내용이 그대로 복원되어야 함. 이렇게 중지시에 저장되고 속개시에 복원되는 프로세스의 실행에 필요한 모든 정보를 프로세스의 문맥(context)이라 함. 문맥 교환(Context Switch)이란 하나의 프로세스가 CPU 를 사용 중인 상태에서 다른 프로세스가 CPU 를 사용하도록 하기 위해, 이전의 프로세스의 상태(문맥)를 보관하고 새로운 프로세스의 상태를 적재하는 작업을 말함. 한 프로세스의 문맥은 그 프로세스의 프로세스 제어 블록에 기록됨

비유 3

프로세스간통신 (Inter-Process Communication) : 실행중인 프로세스 간에 데이터를 주고받는 기법

Process 는 완전히 독립된 실행객체. 서로 독립되어있다는 것은 다른 프로세스의 영향을 받지 않는다는 장점. 그러나 서

로간에 통신이 어렵다는 문제. (쓰레드와 비교됨) 프로세스는 커널이 제공하는 IPC 설비를 이용해서 프로세스간 통신을 함. IPC 에는 많은 방법이 있음 대표적으로 Signal, Soket, Message Queue

시그널: 시그널이란 특정 이벤트 발생했을때 프로세스에게 전달하는 신호. 연산오류발생, 자식프로세스의 종료, 사용자의 종료요청등이 있으며 굉장히 작은 값. 인터럽트라고 부르기도 함. 시그널은 여러종류가 있으며 프로그램내에서는 매크로 상수를 사용함. 프로세스가 시그널을 받으면 각 시그널을 처리하는 핸들러 함수가 동작하게 됨. 시그널을 처리할 동안 방해받지 않도록 mask 를 설정하여 해당 시그널은 금지. 핸들러 실행 후 금지된 시그널은 해제

메세지큐: 메세지큐는 프로세스들 간의 데이터통신(메세지 통신)을 위한 IPC. 메세지는 structured data 이기 때문에, 경계구분 가능. 그러므로 하나의 메세지큐 안에 여러개의 메세지를 버퍼링 할 수 있음. 또한 FIFO, LIFO(Last In First Out) 을 모두 지원하며, 고정된 크기를 가짐. 메세지큐의 구조는 다음과 같음(송신큐 => 메세지큐 => 수신큐) 송신 프로세스가 자기 주소공간에서 메세지큐 주소공간으로 보낼 것을 복사, 메세지큐는 복사된 것을 수신 프로세스의 주소공간으로 복사→ 2 번의 메모리 복사는 큰 부담, 많은 양의 데이터를 전달할 때에는 데이터 대신 그 데이터를 가리키고 있는 포인터만 전달하는 방법을 사용. 메세지큐 사용시에 메세지 송수신을 양방향으로 이루어지게 하면 클라이언트/서버 시스템 설계에 사용가능.

소켓: 떨어져 있는 두 호스트를 연결해 주는 도구로써 운영체제가 만들어주는 SW 적인 장치

비유 4 – 자료조사

TCP flag(URG, ACK, PSH, RST, SYN, FIN)

TCP(Transmission Control Protocol)는 3-WAY Handshake 방식을 통해 두 지점 간에 세션을 연결하여 통신을 시작 하고 4-WAY Handshake 를 통해 세션을 종료하여 통신을 종료 . 이러한 세션연결과 해제 이외에도 데이터를 전송하거나 거부, 세션 종료 같은 기능이 패킷의 FLAG 값에 따라 달라지게 되는데, TCP FLAG 는 기본적으로 6 가지로 구성.

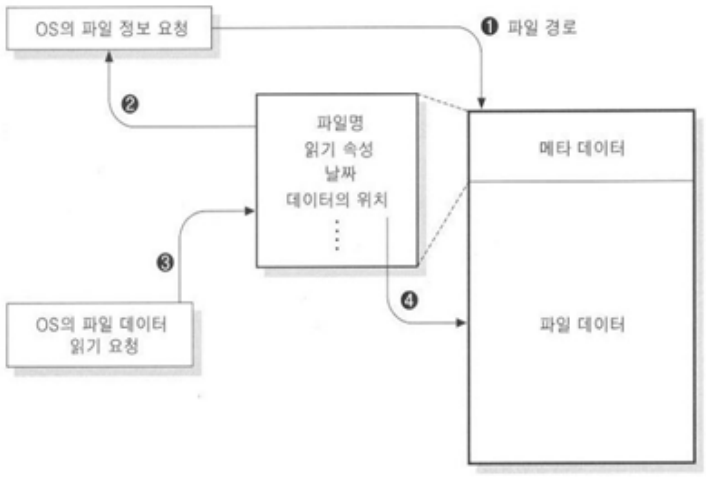
FLAG 순서 : URG | ACK | PSH | RST | SYN | FIN |
(각각 1 비트로 TCP 세그먼트 필드 안에 CONTROL BIT 또는 FLAG BIT 로 정의 되어 있음)

SYN(Synchronization:동기화) – S : 연결 요청 플래그 : TCP 에서 세션을 성립할 때 가장먼저 보내는 패킷, 시퀀스 번호를 임의적으로 설정하여 세션을 연결하는 데에 사용되며 초기에 시퀀스 번호를 보내게됨

ACK(Acknowledgement) – Ack : 응답 : 상대방으로부터 패킷을 받았다는 걸 알려주는 패킷, ACK 응답을 통해 보낸 패킷에 대한 성공, 실패를 판단하여 재전송 하거나 다음 패킷을 전송

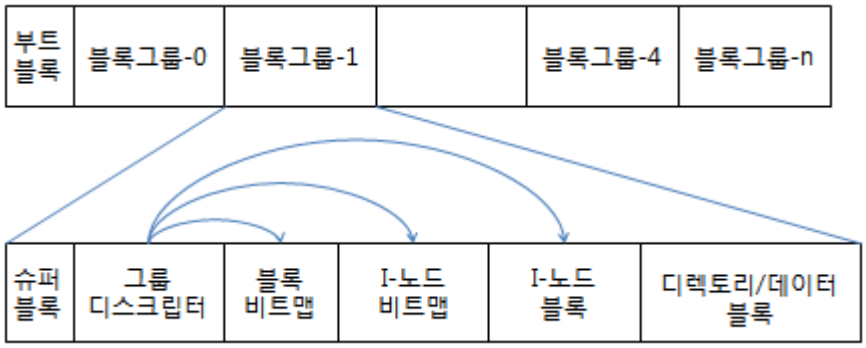
FIN(Finish) – F : 연결 종료 요청 : 세션 연결을 종료시킬 때 사용되며 더이상 전송할 데이터가 없음을 나타냄

비유 5 - 자료조사



파일 시스템(file system, 파일체계)은 컴퓨터에서 파일이나 자료를 쉽게 발견 및 접근할 수 있도록 보관 또는 조직하는 체제. OS 와 HW 사이에 위치하며 윈도우의 경우 FAT(파일정보의 관리: Linked list), NTFS(파일 정보의 관리:B-Tree), 리눅스의 경우 EX3 이 있음.

파일시스템(EX2)의 구조는 다음과 같음



부트 블록(boot block) :파일 시스템으로부터 UNIX 커널을 적재시키기 위한 프로그램이 저장

슈퍼 블록(super block) : 파일 시스템을 기술하는 정보를 저장 → 아이노드, 데이터블록 관련

아이노드(inode) : 파일이나 디렉터리에 대한 모든 정보를 가지고 있는 구조체이다.

데이터 블록: 실제 데이터가 파일의 형태로 저장되는 공간

ext2 파일시스템은 부트섹터(Boot Sector)와 여러 개의 블록 그룹(Block Group)으로 구성. 블록이란 파일시스템에서 데이터를 저장하는 단위이며, 메모리에서 I/O 작업을 한 번 거칠 때 읽거나 쓰는 단위. (파일시스템을 생성할 때 1KB~4KB 사이에서 블록의 크기를 지정 가능), 파일 시스템의 전체적인 정보는 슈퍼 블록(Super Block)과 그룹 디스크립터 테이블(Group Descriptor Table)에 저장. 슈퍼 블록과 그룹 디스크립터 테이블은 0 번 블록 그룹의 정보만을 사용하지만 주요 데이터이기 때문에 손상될 경우를 대비하여 모든 블록 그룹에 사본이 저장되어 있음