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

<리눅스 시스템 프로그래밍>

2018.03.20-19일차

강사 – 이상훈
gcccompil3r@gmail.com

학생 – 안상재
sangjae2015@naver.com

1. 리눅스 커널에서 함수 포인터를 사용하는 이유

리눅스는 어떤 파일시스템도 지원하는 가상파일시스템을 가지고 있다. (그로 인해 리눅스는 윈도우 감지 가능, 윈도우는 리눅스 감지 불가.) 리눅스가 모든 파일 시스템을 지원하고 각 함수의 인자가 매우 많기 때문에 모든 것을 다 외우고 프로그래밍을 할 수는 없다. 그래서 함수 포인터라는 체계를 도입해서 효율적으로 사용할 수 있게 커널 내부에 정의되어 있다.

- 리눅스는 어떤 파일시스템도 지원하기 때문에 리눅스에서 윈도우를 감지할 수 있지만, 윈도우에서는 2개의 파일시스템(FAT, NTFS)만을 지원하기 때문에 리눅스를 감지하지 못한다.

- 프로그램 1개당 task_struct이 1개가 있다. task_struct의 file_struct안에 file * 가 파일의 주소를 가리킨다. open을 2개 하면 별도의 파일디스크가 생긴다. 그러므로 한 파일에서 포인터가 이동한다고 해서 다른 파일의 포인터는 이동하지 않는다.

2. 유닉스의 철학 : 모든 것을 파일로 표현한다. (여기서 파일에는 장치 파일, 파이프, 디렉터리, 네트워크 등 모든 것이 포함된다.)

3. 파일은 읽거나 쓰기 전에 반드시 열어야 한다. 시스템 콜에 의해 파일을 열고 닫고, 읽고 쓰게됨.

1) open() 파일 열기 : int open(const char *name, int flags, mode_t mode)

- 이름이 name인 파일을 file descriptor (fd)에 맵핑하고, 성공하면 fd를 반환한다. 실패하면 -1을 반환함.

- flags 인자는 O_RDONLY(읽기 전용), O_WRONLY (쓰기 전용), O_RDWR (읽기/쓰기) 가 들어감. OR 연산을 통해 동시에 사용할 수 있음.

2) read() 읽기 : ssize_t read (int fd, void *buf, size_t len)

- read()가 호출되면 fd가 참조하는 파일의 현재 파일 오프셋에서 len 바이트만큼 buf에 저장한다. 실패하면 open()과 마찬가지로 -1을 반환한다. 파일 오프셋은 fd에서 읽은 바이트 크기만큼 이동한다.

3) write() 쓰기 : ssize_t write (int fd, const void *buf, size_t count)

- count 바이트만큼 fd가 참조하는 파일의 현재 파일 위치에 시작지점이 buf인 내용을 기록한다. 반환값은 쓰기에 성공한 바이트 수를 반환하고, 에러가 발생하면 -1을 반환.

4) lseek() : off_t lseek (int fd, off_t pos, int origin)

- lseek()는 파일의 시작, 혹은 끝 지점으로 오프셋을 이동하거나, 현재 파일의 오프셋을 알아내는 데 가장 많이 사용된다.

-SEEK_CUR : fd의 파일 오프셋을 현재 오프셋에서 pos를 더한 값으로 설정

SEEK_END : fd의 파일 오프셋을 현재 파일 크기에서 pos를 더한 값으로 설정

SEEK_SET : fd의 파일 오프셋을 pos 값으로 설정.

* quiz1

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <fcntl.h>

int extract_idx;

typedef struct __queue
{
	int data;
	struct __queue *link;
} queue;

bool is_dup(int *arr, int cur_idx)		// 중복 체크 함수
{
	int i, tmp = arr[cur_idx];

	for(i = 0; i < cur_idx; i++)
		if(tmp == arr[i])
			return true;

	return false;
}

void init_rand_arr(int *arr, int size)	// arr배열에 1~10 숫자를 임의로 저장하는 함수.
{
	int i;

	for(i = 0; i < size; i++)
	{
redo:
		arr[i] = rand() % 10 + 1;

		if(is_dup(arr, i))
		{
			printf("%d dup! redo rand()₩n", arr[i]);
			goto redo;
		}
	}
}

void print_arr(int *arr, int size)  // 배열의 데이터를 출력함.
{
	int i;

	for(i = 0; i < size; i++)
		printf("arr[%d] = %d₩n", i, arr[i]);
```

```
}


queue *get_queue_node(void) // 노드 생성
{
        queue *tmp;
        tmp = (queue *)malloc(sizeof(queue));
        tmp->link = NULL;
        return tmp;
}

void enqueue(queue **head, int data) // queue에 노드를 추가함.
{
        if(*head == NULL)
        {
                *head = get_queue_node();
                (*head)->data = data;
                return;
        }

        enqueue(&(*head)->link, data);
}

void extract_even(queue *head, int *extract)   // queue에서 짝수 data를 배열에 저장함.
{
        queue *tmp = head;

        while(tmp)
        {
                if(!(tmp->data % 2))
                        extract[extract_idx++] = tmp->data;
                tmp = tmp->link;
        }
}

int main(void)
{
        int i, fd, len, sum = 0;
        char *convert[10] = {0};
        int arr[11] = {0};
        char tmp[32] = {0};
        int extract[11] = {0};
        int size = sizeof(arr) / sizeof(int) - 1;
        queue *head = NULL;

        srand(time(NULL));

        init_rand_arr(arr, size); // 배열 데이터 초기화
        print_arr(arr, size);      // 배열 데이터 출력

        for(i = 0; i < size; i++)
                enqueue(&head, arr[i]);         // 배열을 queue에 삽입함.
```

```
	extract_even(head, extract);	// queue의 데이터 중 짝수 데이터를 extract 배열에 빼옴.
	printf("₩nExtract:₩n");
	print_arr(extract, extract_idx);	// extract 배열 출력

	fd = open("log.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644); // "log.txt" 이름의 파일을 생성, 읽기 전용, 지우기 옵션으로 open함.

	for(i = 0; i < extract_idx; i++)
		sum += extract[i];	// extract배열의 짝수 데이터를 모두 더함.

	sprintf(tmp, "%d", sum);	// sum을 출력하고 tmp 버퍼에 sum을 저장한다.
	write(fd, tmp, strlen(tmp));	// fd파일에 tmp 데이터를 write함.
	close(fd);			// fd파일을 close함.

	return 0;
}
```