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

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
36-37 일차 (2018. 04. 12-13)

# 목차

# Chapter 6 : 인터럽트와 트랩 그리고 시스템 호출

## 1) 인터럽트 처리 과정

인터럽트란 주변 장치와 커널이 통신하는 방식 중 하나이다. **주변장치나 CPU 가 자신에게 발생한 사건을 리눅스 커널에게 알리는 매커니즘**이다. 인터럽트가 발생되면 운영체제는 왜 인터럽트가 발생했는지 살펴보고 적절한 작업을 처리해야 한다. 이 작업을 '**인터럽트 핸들러(interrupt handler)**'라는 함수가 처리한다. 인터럽트가 생기면 하던 작업을 중단하고 인터럽트를 실행해야 한다.

시스템이 운영되는 도중 발생되는 인터럽트는 원인데 따라 크게 '외부 인터럽트'와 '트랩' 2 가지로 구분할 수 있다.

- 외부 인터럽트 : 현재 수행중인 태스크와 관련 없는 주변장치에서 발생된 비동기적인 하드웨어적인 사건
- 트랩 : 소프트웨어적인 사건. 예외처리(exception handling)라고도 한다. 0 으로 나누는 연산(devide_by_zero), 세그멘테이션 결함, 페이지 결함, 보호 결함, 시스템 호출 등이 있다.

즉, 인터럽트의 발생 기준은 **CPU 내부에서 감지하는지, 외부에서 감지하는지**에 따라 분류한다. 그렇다면 인터럽트가 발생하면 어떻게 운영체제의 인터럽트 핸들러가 수행되는 것일까? 이는 모든 CPU 가 인터럽트가 발생하면 program counter(또는 instruction pointer) 레지스터의 값을 미리 정해진 특정 번지로 변경하도록 정해져 있기 때문이다. 예를 들어 ARM 계열은 인터럽트가 발생하는 순간 0x00000000+offset 번지로 점프한다. offset 은 인터럽트의 종류에 따라 결정되며, reset 인터럽트는 offset 이 0(리셋이 발생되면 0 부터라는 뜻)이고 underfined instruction 은 4, software interrupt 는 8 이다. 그러나 실제로는 인터럽트 핸들러의 용량이 4byte 보다 크기 때문에 실제 0x00000000 에 reset 인터럽트 핸들러를 기록하는 대신, 다른 위치에 작성해 두고 그곳으로 점프하는 명령어만을 기록해두는 방식을 쓴다. 이런 처리방식을 외부 인터럽트와 트랩은 동일하게 사용한다. 외부 인터럽트와 트랩을 처리하기 위한 루틴을 함수로 구현해 놓고, **각 함수 시작 주소를 리눅스의 IDT 인 idt_table** 이라는 배열에 기록한다. 0~31 까지의 32 개의 엔트리를 CPU 의 트랩 핸들러를 위해 할당하고 그외의 엔트리는 외부 인터럽트의 핸들러를 위해 사용한다. 이 때, 외부 인터럽트를 발생시킬 수 있는 라인은 한정된 개수를 가진다. 따라서 **외부 인터럽트를 위한 번호는 별도로 irq_desc 테이블**을 통해 관리 한다. 32~255 까지의 idt_table 에는 같은 인터럽트 핸들러 함수가 등록되어 있으며, 이 함수는 'do_IRQ()'를 호출한다. do_IRQ()는 '외부 인터럽트' 번호를 가지고 irq_desc 테이블을 인덱싱 하여 해당 '외부 인터럽트'번호의 irq_desc_t 자료구조를 찾는다. 이 자료구조에는 하나의 인터럽트를 공유할 수 있도록 action 이라는 자료구조 리스트가 있다. 이 리스트는 단일 인터럽트 라인을 공유하는 것이 가능하다. 여기서 128 번만 제외하는데 이는 별도로 128 번에는 유일한 '트랩', 소프트웨어 인터럽트(system call 명령어)가 있기 때문이다.

이런 핸들러의 호출은 사실 **커널이 인터럽트를 받으면 즉시 인터럽트 핸들러를 호출할 수 있는 것은 아니고, context switching 을 해야 한다**. 인터럽트 발생 전, Task 문맥 저장을 하기 위해서이다. 인터럽트 처리가 완료 되면 문맥을 복원하는 작업등을 수행한다. 즉, 핸들러가 인터럽트 작업을 마친 후, 그 전 작업을 하던 저장위치에서 시작한다는 것이다.

### 2) 시스템 호출 처리과정

이는 C Library(커널 소스)를 봐야 한다. 시스템 호출이란 사용자 수준 응용 프로그램들에게 커널이 자신의 서비스를 제공하는 인터페이스다. 따라서 사용자가 운영체제의 기능이나 모듈을 활용하기 위해서는 반드시 시스템 호출을 사용해야 한다. 시스템 호출은 커널로의 진입점이라고 볼 수 있다. sys_fork(), sys_read(), sys_nice()가 그 예이다. 시스템 호출 처리 과정은 다음과 같다.

1. 사용자가 fork()라는 system call 을 요청(가정)한다.
2. fork()라는 이름의 라이브러리가 호출(C Library 구현)된다. 사용자 대신 트랩을 요청하는 것이다.
3. fork()에 할당된 고유 번호 2 를 eax 레지스터에 넣고 0x80 을 인자로 트랩을 건다.
4. 커널은 context swhiching 을 하며, 트랩 번호(0x80)에 대응되는 엔트리에 등록된 함수를 호출한다.
5. eax 의 값을 인덱스로 sys_call_table 을 탐색하여 sys_fork()의 함수포인터를 얻는다.

→ 결국 사용자 수준 응용이 fork()라는 시스템 호출을 요청했으며, IDT 테이블과 sys_call_table 을 이용해 커널에서 구현 된 sys_fork()함수가 호출되는 것이다.

## Chapter 7 : 리눅스 모듈 프로그밍

리눅스에서는 동적 커널 모듈 또는 간단히 모듈이라고 불리는 메커니즘을 제공한다. 커널 대부분의 기능은 모듈로 구현될 수 있다.

### 1) 마이크로 커널

리눅스는 커널구조상 모노리틱(모놀리식, monolithic) 커널이다. 이는 커널이 제공해야 할 모든 기능 즉 태스크 관리, 메모리 관리, 파일시스템, 디바이스 드라이버, 통신 프로토콜 등의 기능이 단일한 커널 공간에 구현된 구조이다. 이 모놀리식 커널과 구조상으로 반대되는 개념이 마이크로 커널이다. 마이크로 커널은 커널 공간에 반드시 필요한 기능들만 구현하는데, 문맥 교환, 주소 교환, 시스템 호출 처리, 디바이스 드라이버의 일부 등 **하드웨어와 밀접하게 관련된 기능들을 커널 공간에 구현**한다. 이 기능은 **탈부착이 가능**하여, 리눅스는 기본적으로 모놀리식을 사용하지만 디바이스 드라이버만큼은 마이크로 방식을 사용한다. 디바이스 드라이버만큼은 탈부착이 가능해야 하기 때문이다. 그래서 리눅스가 하이브리드 방식인 것이다.

이 마이크로 커널의 장점은 무엇일까? **커널의 크기를 작게** 할 수 있다. 이 장점으로 인해

1. 커널 소스도 작고 깨끗해질 수 있어 관리, 개선, 유지 등이 쉬워진다.
2. 커널 크기가 작아서 PDA , 휴대폰 , 노트북 등 휴대용에 많이 쓰인다.
3. 많은 기능이 사용자 공간에서 서버 형태로 구현되기 때문에 분산 환경인 클라이언트/서버 모델에 잘 적용이 된다.

이 때문에 최근에 개발되는 운영체제들은 대부분 마이크로 커널 구조를 갖는다.

## Chapter 8 : 디바이스 드라이버

### 1) 디바이스 드라이버 일반

유닉스 계열 시스템에서 **모든 것은 파일로 취급**된다. 따라서 사용자 태스크는 현재 접근하려는 파일이 모니터인지 키보드인지 정규파일인지 신경 쓰지 않고 그저 파일만을 접근하면 된다. 이는 open(), read(), write(), close() 등의 일관된 함수 인터페이스를 통해서 모니터, 키보드, 정규파일 등

의 파일을 접근하는 것이 가능하다는 것이다. 다만, 장치를 가르키는 파일을 정규파일과 구분하기 위해 장치파일이라 부른다.

가. 사용자 입장에서 디바이스 드라이버

사용자 태스크가 접근하는 장치 파일이라는 개념은 VFS 가 제공하는 파일 객체를 의미한다. 파일 객체에 사용자 태스크가 행할 수 있는 연산은 struct file_operations 라는 이름으로 정의되어 있다. 사용자 태스크가 **file_operations 구조체에 정의되어 있는 함수를 통해 장치파일에 접근**할 때, **호출할 함수를 정의하고 구현**해주는 것이 '**디바이스 드라이버**' 이다. 이는 사용자 태스크는 디바이스 드라이버 개발자가 작성한 여러 가지 함수들을 일일이 알 필요 없이 파일 객체에 정의되어 있는 함수를 호출함으로써 장치에 접근할 수 있다는 장점이 있는 것이다.

리눅스는 시스템에 존재하는 여러 개의 디바이스 드라이버를 구분하기 위해 각 디바이스 드라이버마다 고유한 번호를 정해준다. 이 고유한 번호를 주 번호(Major number)라 한다. 리눅스에선 각 장치는 자신을 나타태는 장치 파일을 가지며, 이 장치 파일을 관리하는 아이노드 객체에 주 번호가 기록되어 있다. 구제척으로 **아이노드 객체에는 i_rdev 필드가 존재하는데 이 i_rdev 필드에 주 번호(장치 종류)와 부 번호(장치 개수)를 저장**한다. 주 번호는 12bit 가 사용(4096 개를 지원하므로)되며, 부 번호는 20bit 가 사용된다. 주 번호를 알게 되면, 장치파일에 등록된 디바이스 드라이버 내부 함수를 호출할 수 있게 된다.

나. 개발자 입장에서 디바이스 드라이버

## 2) 문자 디바이스 드라이버 구조

새로운 디바이스 드라이버를 구현할 때, 필요한 작업 단계가 있다.

1. 디바이스 드라이버의 이름과 주 번호를 결정해야 한다.
2. 디바이스 드라이버가 제공하는 인터페이스를 위한 함수들을 구현해야 한다. 그리고 이 함수들의 시작 주소는 파일 연산이라는 자료구조에 초기화 되어야 한다.
3. 새로운 디바이스 드라이버를 커널에 등록해야 한다.
4. /dev 디렉터리에 디바이스 드라이버(장치를 구동시킬 수 있는 소프트웨어)를 접근할 수 있는 장치파일을 생성해준다.