Linux Kernel

2022.07.13

1. 리눅스 커널 내부 구조 (By directory)

A. kernel: Task 관리자가 구현되어 있으며, task의 생성 및 소멸, 실행, 스케줄링, 시그널 핸들링이 구현되어 있음.

B. arch: 리눅스 커널 중 하드웨어에 dependent한 부분들이 구현되어 있음. Context Switch 또한 cpu register 값을 넣는 과정이 있으며, 이는 architecture에 따라 다르므로 arch에 구현되어 있는 것으로 추측. 이 directory는 cpu 제조사에 따라 하위 계층으로 나누어져 있음. 제조사에 따라 bootstrap code, context switch, page fault handling등이 구현되어 있음.

C. fs: 리눅스는 다양한 파일시스템을 지원한다. 예를 들어, 윈도우는 주로 NTFS, FAT 계열의 파일시스템을 사용하지만, 리눅스는 Ext4를 사용한다. 리눅스는 이러한 환경에 대응하기 위해 다양한 파일시스템이 지원되도록 구현하였으며 이는 fs 디렉토리에 구현 되어있다. 또한, 다양한 파일시스템 지원 방법으로 VFS(Virtual File System)을 사용하는데, 이것 또한 fs 디렉토리에 구현되어 있다.

D. mm: 메모리 관리가 구현되어 있다. Physical memory, Virtual memory management와 task마다 할당되는 virtual memory object management가 구현되어 있다.

E. driver: 리눅스에서 다양한 디바이스들을 지원하기 위해 구현되어 있는 드라이버들이 있는 디렉토리이다. 디바이스마다 다른 환경에 대응하기 위해 드라이버를 만들고 이를 통해 사용자는 해당 디바이스의 특정된 함수를 알 필요 없이 일반적인 함수를 사용하면 파일시스템과 드라이버가 자동으로 변환하여 원하는 작업을 해준다. 크게 character device driver, block device driver, network device driver로 구분되어 진다.

F. net: 리눅스가 지원하는 통신 프로토콜이 구현되어 있는 디렉토리이다. 리눅스 커널에서 상당히 많은 부분을 차지한다. 네트워크를 추상화하여 사용자에게 제공하는 소켓 인터페이스도 net에 구현되어 있다.

윈도우와 달리 리눅스는 버젼을 업그레이드 할 때 설치되어 있는 os를 완전히 삭제하고 다시 설치하지 않는다. 터미널에서 자신이 설치하고자 하는 커널 버젼을 검색하고 이를 apt를 이용하여 kernel을 설치하고 reboot을 하면 적용된다. 요즘 윈도우 또한 이러한 기조로 새 버젼을 출시하려는 것처럼 보인다. 실제로 우분투 커널 버전을 업그레이드 해보았는데, 바로 적용되지는 않고 Grub에서 해당 버전을 선택해주는 과정이 필요하였다. 추가로, 커널 버전을 업하였을 때, 네트워크 및 디스플레이 디바이스 드라이버가 자동으로 잡히지 않는 현상이 발생하기도 하였다.

2. Task scheduling

- Hyper Threading: Multi-core와 SMP system에서 한 코어의 run-queue에 task들이 몰려있다면 이를 다른 코어의 run-queue로 이주시켜 load_average를 맞춰준다. 이때, 리눅스가 부팅 되고 나서 인식하고 있는 구조를 기반으로 cache를 이용한 성능 향상을 위하여 이주시킬 코어의 우선순위를 정해 해당 코어가 적합한지 판별한다. 이를 통해 성능 개선을 이끌어낸다.

- CFS & {RR, FIFO, DEADLINE} : 모두 task scheduling 알고리즘인데, 두 분류로 나눈 이유는 전자는 일반 task로 분류한 task들을 scheduling 할 때 쓰이는 알고리즘이고, 후자는 실시간 task로 분류한 task를 scheduling 할 때 쓰이는 알고리즘이다. 여기서 처음 접한 알고리즘은 CFS, DEADLINE이다.

먼저 DEADLINE은 알고리즘 수업에 배운 Earliest deadline first algorithm에서 착안한 것으로 보이며, 실제로 이 알고리즘은 비슷한 4가지 중에서 유일하게 optimal 했던 것으로 기억한다. DEADLINE의 요점은 가장 급한 task부터 처리한다는 것이다. 따라서, current time + running time < DEADLINE이어야 한다. 이 알고리즘은 Red-Black tree를 사용하기 때문에 task scheduling은 O(1)에 가능할 것으로 보이며, 또한, task가 새로 들어온다고 해도 O(log n)의 time-complexity를 가질 것이므로 O(n) 보다 Optimal하다.

다음은 CFS이다. CFS는 Completely Fair Scheduling의 약자이고, task들의 CPU 점유 시간이 1:1로 항상 동일해야 하다는 것이다. 여기서 특이한 점은 보통 점유 시간이 1:1이라 하면 막연하게 1초, 1초라 생각할 수 있는데, CFS는 시간 단위당 그 안에서 비율이 동일해야 한다. 따라서, 1초가 시간 단위고 두 task가 있다면 각각 task의 점유시간은 0.5초 / 0.5초이다. 이렇게 된다면, Round Robin처럼 optimal한 단위 시간을 찾는 것이 중요할 것이다. 너무 길면 뒤의 task들이 기다리는 시간 또는 다 끝내도 기다리는 시간이 늘어날 수 있으며, 너무 짧으면 Context-switch가 빈번하게 일어나 Overhead가 많아지기 때문이다.

3. Memory Management

- Node & Zone : 리눅스에서 CPU가 메모리에 접근 할 때, 접근 속도가 동일한 메모리들을 Bank라 한다. UMA 구조에서는 이 bank가 하나이고, NUMA 구조에서는 bank가 다수개인데 리눅스에서 이 bank를 나타내는 구조가 node이다.

일부 ISA 버스 기반 디바이스의 경우 정상적인 동작을 위해서 반드시 물리 메모리중 16MB 이하 부분을 할당해 줘야 하는 경우가 있다. 따라서 노드에 존재하는 물리메모리 중 16MB 이하 부분은 특별하게 관리하는데, 이를 위해 node 의 일부분을 따로 관리할 수 있도록 자료구조를 만들어 놓았다. 이를 zone 이라 부른다.

- Buddy Allocator & Slab Allocator

➔ Linux에서 4KB를 보통 Page size로 정하는데, 항상 프로세스들이 4KB만 요구하는 것은 아니다. 만약, 요청한 size가 4KB가 작다면 이를 그대로 1 page를 주면 internal fragmentation이 발생해 낭비가 심할 것이다. 이 경우에는 Slab allocator를 사용한다.

➔ 요청한 size가 4KB보다 크다면 여러 page를 주면 되겠지만, memory management의 용이성과 external fragmentation을 줄이기 위해 Buddy allocator가 사용되며 이를 보완한 Lazy buddy allocator가 사용된다.

1	2	3	4	5	6	7	8
0		1		1		0	
0				1			
0							

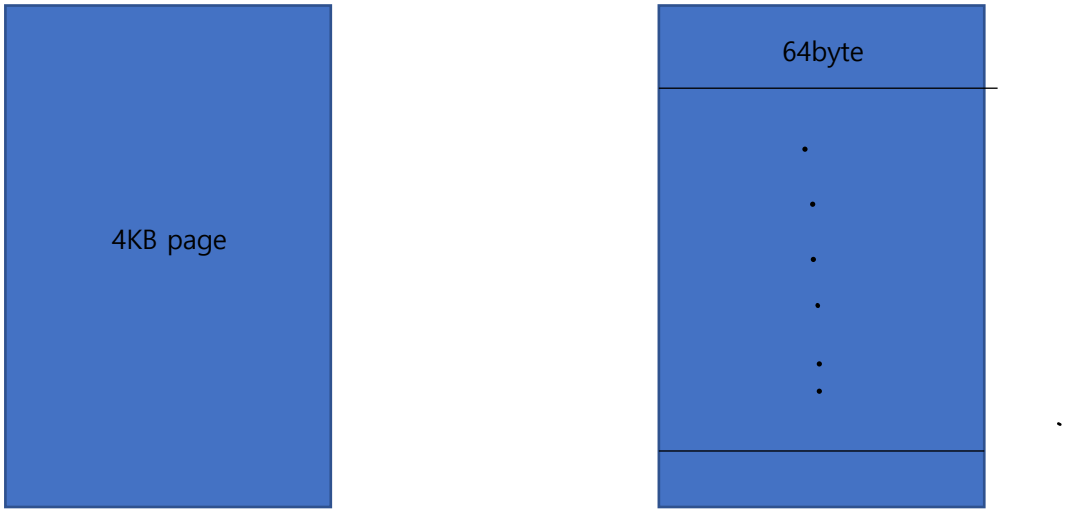
<Buddy allocator>

위의 표에서 가장 상단 줄은 frame을 나타낸다. 빨간색으로 음영 처리 되어 있는 부분은 frame을 process가 점유하고 있는 것이다. 그 밑으로는 order-N에 따라서 숫자를 매길 수 있다. 여기서 N은 0부터 시작하며 2^N개의 frame을 보는 것이다. 두 frame의 상태가 같으면 0, 다르면 1이다. 예로, 두번째 줄은 Order – 0이므로 1개의 frame씩 기준으로 보게 되며, (1,2)는 frame이 occupied되어 있으므로 0, (3,4)는 각각 다른 상태이므로 1 이러한 기준으로 숫자를 매긴다.

이 상태에서 3개의 frame을 요청했다면 요청된 frame을 만족시킬 수 있는 최소 order의 상태를 먼저 확인하고, 할당이 가능하다면 이 level에서 할당을 완료하고, 만약 할당이 불가능하다면 상위 level에서 가능한 부분을 반으로 나누어 한쪽을 할당하고 한쪽은 하위 level available에 count한다. 이때 나누어진 두 부분을 서로의 buddy라고 일컫는다.

만약 하나의 frame이 할당 & 해제를 계속 반복한다면 어떻게 될까? 그 level에서 해결이 가능하다면 숫자를 바꾸는 overhead 정도만 발생하겠지만, 만약 해당 level에서 해결이 불가능하여 상위 level의 frame에서 반으로 나누어 할당을 한다면, 해제시에는 이를 다시 합쳐주는 과정이 필요하다. 이 과정이 계속 반복된다면 꽤나 큰 overhead일 것이다. 이에 Linux는 kernel version 2.6.19부터 Lazy buddy allocator를 적용한다. Cache스러운 면모를 적용하여 특정 패턴을 가지는 allocation은 merge되는 것을 뒤로 미룬다 해서 Lazy라는 명칭이 붙었다. 이로 인해 zone에 available한 memory가 증가하였다. Merge process는 Loop을 돌리면서 buddy가 합쳐질 수 있는지 확인하고 가능하다면 현재 order의 free frame수를 줄이고, 상위 order의 free frame수를 늘린다.

<Slab Allocator>



만약 요청된 size가 64byte인데 4KB size를 모두 주는 것은 낭비가 될 것이다. 이에 따라, linux에서 slab allocator는 어떠한 한 페이지를 특정 크기로 나누어 놓고, 4KB보다 작은 size의 요청이 들어오면 이곳에서 할당을 진행한다. 이 때, size는 Frequently하게 할당 및 해제가 되는 size로 정한다. 오른쪽 page의 상태를 나타낼 때 모든 Object가 할당 가능하면 free, 모든 object는 아니라면 partial, 모든 object가 할당 불가능하면 full이라는 상태를 붙인다.

추가로, 해당 단원에서 메모리를 관리하는 구조체를 언급하였는데, 이에 대해 좀더 알아보는 것이 필요하다고 생각한다. 해당 구조체는 mm_struct, vm_area_struct이다. 특히, vm_area_struct에서 동일한 속성의 영역이라면 이를 합친다고 말하였는데, code는 code끼리 묶고, stack은 stack끼리 묶는다는 것인지, 아니면 다른 의미가 있는지 아직 파악하지 못하였다. 뭔가 위에서 이야기 한 것처럼 하면 문제가 발생할 것 같아서 알아보는 것이 필요하다.

4. Interrupt

- Definition : Device들이 kernel에게 어떠한 사건이 발생하였음을 알리는 것.

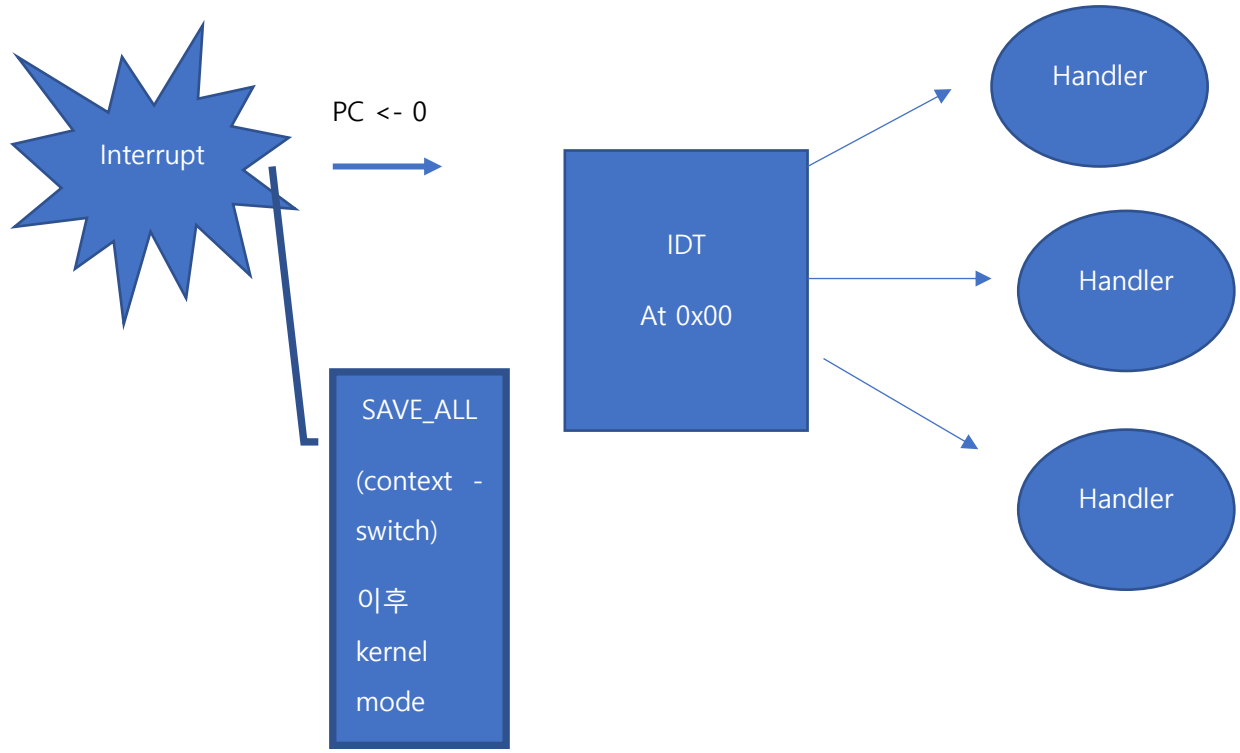
- Interrupt의 종류 :

  Asynchronous interrupt : 현재 작동하고 있는 프로세스에 의해 발생되지 않고 외부의 요인으로 인해 발생한 interrupt (timer expiration, I/O..)

  Synchronous interrupt (trap) : 작동하고 있는 프로세스에 의해 동기적으로 발생한 interrupt (divide by zero, segmentation, page fault, system call)

➔ 리눅스에서는 두 interrupt 모두 동일한 방식으로 처리함.

<Process of interrupt>



위 과정에서 handler의 경우 각 interrupt cause에 대응하여 번호가 배정되어 있다. 0 ~ 31은 trap, 32 ~ 부터는 asynchronous interruput을 위한 PIC가 잡혀있다. 또한, trap의 경우 처리 이후 PC 진행 방식에 따라 3가지로 분류된다.

	진행 방식
fault	PC <- **PC**
trap	PC <- **PC+4** (PC <- PC)
abort	Critical issue -> **terminated**

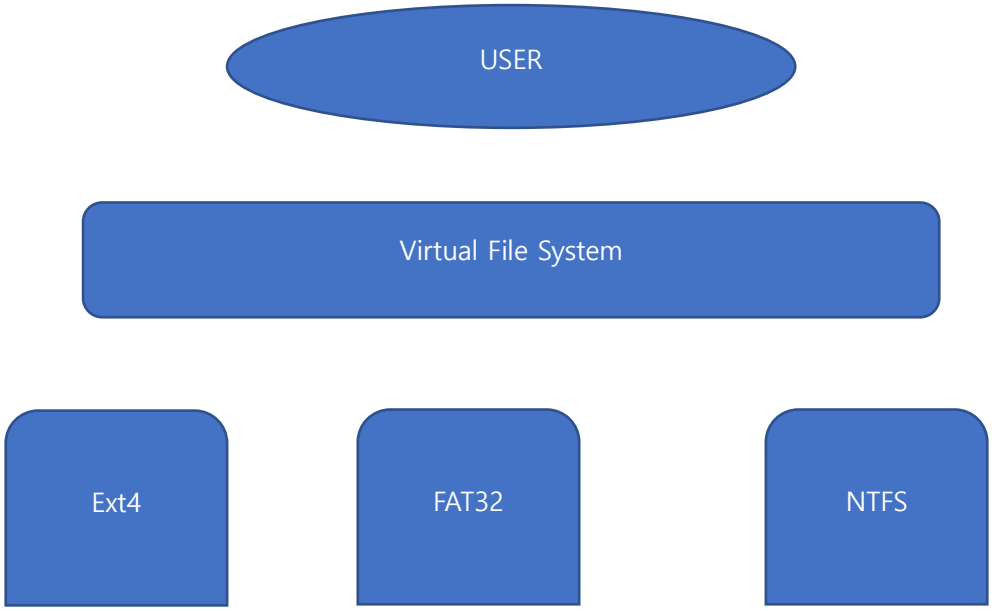
디바이스 드라이버 설계를 고려할 때, interrupt 말고 polling과 같은 것을 적용하면 안되는지 궁금해졌다. 자주 일어나면서 처리 속도가 빨라야 하는 상황에서는 interrupt보다는 polling이 나을 수도 있다는 생각을 하였기 때문이다. Interrupt의 경우 context-switch와 kernel / user mode switch라는 overhead가 발생하는데, 이 overhead는 상당히 큰 편에 속하므로 자주 발생하는 경우에는 interrupt는 성능 저하가 예상된다. 또한, polling은 계속해서 확인하므로 즉각적인 반응이 가능하다. 하지만, 이 점에서도 분명 penalty는 존재할 것이다.

5. File System

- Virtual File System(VFS)

현재 파일시스템은 다양하게 개발되어 있다. 리눅스는 Ext2,3,4, 윈도우는 NTFS 등 다양한 파일시스템이 존재하는데, 각각 파일시스템마다 함수들이 구현되어 있다. 그런데, 사용자가 파일을 다룰 때 각각 파일마다 해당하는 파일시스템을 파악하고 이에 따라 함수를 부르기에는 너무 번거롭다.

우리는 객체 지향 프로그래밍을 배우면서 polymorphism이라는 것을 알게 된다. 하나의 추상 클래스를 두고, 그 밑에 비슷한 기능을 하는 클래스를 두는 것이다. 리눅스는 이러한 concept을 파일시스템에 접목시켰고 이것이 VFS이다.



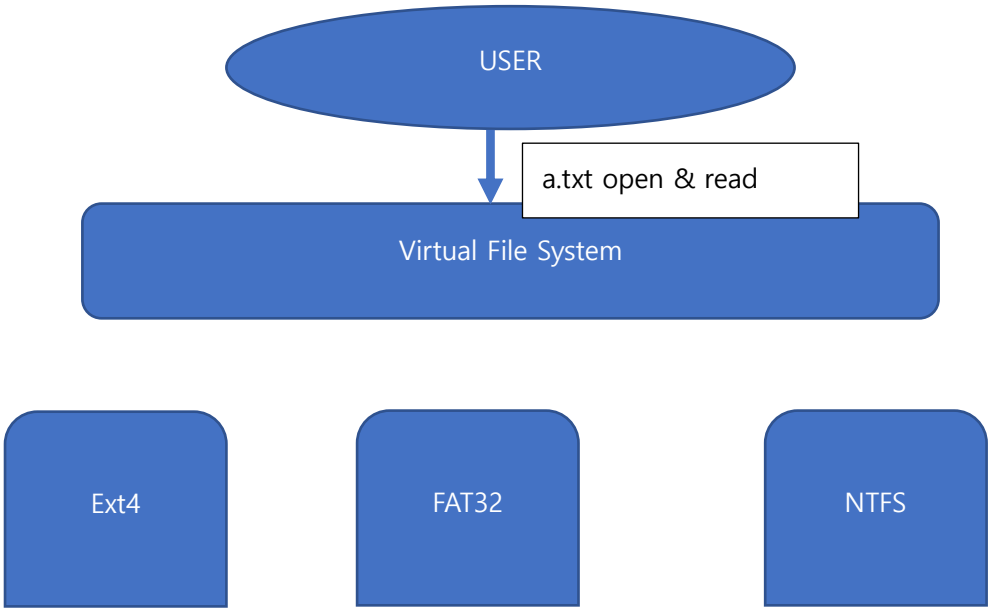
위와 같은 그림에서 VFS layer가 존재하지 않는다면 사용자는 각각의 파일시스템에 직접 접근해야 하기 때문에 번거로울 것이다. 여기에 VFS layer라는 가상화된 layer를 하나 넣어, 이 layer에서 필요한 작업을 하는 일종의 interface를 추가시키는 것이다. 이 필요한 작업을 하기 위해 VFS에는 4개의 object가 존재한다.

- Super block : 슈퍼 블록은 각 파일시스템마다 하나씩 주어진다. 각 파일시스템마다 하나씩 주어지며 각각의 파일시스템에 대한 정보를 저장해둔다.
- Inode : inode 객체는 파일시스템에서 파일에 대한 정보를 가져오면 이에 저장하는 역할을 한다. Msdos 파일 시스템이었다면 해당 파일의 디렉터리 엔트리를, ext2라면 디렉터리 엔트리와 inode에 대한 내용을 가져워서 VFS inode에 넣는다.
- File : 해당 객체는 task들이 아이노드 객체에 접근 할 때만 메모리에 유지된다. 즉, 여러 task들이 하나의 파일에 접근하더라도, 각 task들이 접근하고자 하는 세부적인 내용은 다를 수 있으므로 이에 대한 offset과 같은 정보를 저장한다. 또한, 몇개의 task들이 해당 파

일을 열람하고 있는지에 대한 count도 있을 것이다. (추측)

- Dentry : Task가 파일에 접근하려면 VFS의 Inode와 연결되어야 하는데 이를 빠르게 하기 위해 일종의 cache역할을 한다.

<VFS system call process>



User가 a.txt라는 파일을 open하고 read하는 요청을 보냈다고 하자. VFS는 이 파일이 어떠한 파일시스템의 파일인지 파악하고, 이 파일시스템에 대한 슈퍼블록을 획득한다. 이 과정에서 f_op이라는 구조체 또한 얻어온다. F_op 구조체는 file_operation의 구조체로 각 파일시스템의 operation을 가리키고 있는 함수 포인터들의 구조체이다. 즉, 이후 read / write와 같은 system call을 user에서 보내면 이 system call은 VFS의 f_op에서 함수 포인터를 거쳐 해당 파일 시스템의 함수를 실행하게 된다. 얻어온 정보를 inode에 채워넣고 이에 대한 data를 User에게 최종적으로 넘겨 주는 것이다.

이 구조는 후의 device driver와 깊은 연관이 존재한다. Device driver 또한 사용자에게 device를 다룰 수 있는 일종의 interface를 추상화하여 제공하는 것이기 때문에 VFS와 공통점이 존재하며 VFS에서 driver file까지 관리하기 때문에 VFS 또한 필요하다.

6. Module Programming

Module의 등장배경 ->

Monolithic Kernel : Kernel이 제공해야하는 service들을 한 공간에 모두 구현

Micro Kernel : Kernel이 반드시 필요한 기능만 한 공간에 구현하고 나머지는 분산.

(Context-switch, Address Translation, System Call Handling, Device Driver)

주로 Hardware와 밀접하게 관련된 기능들이 구현되어 있음

➔ 다른 것들은 사용자 공간에 구현하며 커널의 크기가 작아지고, 휴대용 기기 목적으로 한 시스템 OS 구현에 용이, Client-Server 용이

Linux = monolithic Kernel + micro kernel (linux에서는 module 형식 채택)ㄱ

간단한 Module Programming

1. Module source code 작성

2. Makefile 작성 후 .ko 파일 생성

3. insmod로 kernel에 module 등록

4. rmmod로 module제거

<hello_module.c>

```c
#include <linux/kernel.h>
#include <linux/module.h>

int hello_module_init(void){
    printk(KERN_EMERG "Hello Module~!, IN kernel \n");
    return 0;
}
void hello_module_exit(void){
    printk("<0>Bye Module~!\n");
}

module_init(hello_module_init);
module_exit(hello_module_exit);
MODULE_LICENSE("GPL");
```

<Makefile>

```
obj-m += hello_module.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Kernel Print ('dmesg')

dmesg를 터미널에 입력하였을 때 정상적으로 모듈이 등록되고 해제되는 것을 확인하였다.

<Troubleshooting>

1. 여러 헤더 파일 & 소스 파일이 경로에 없다고 나오는 상황

-> 리눅스 소스 파일에서 찾아서 추가시켜 주어도 구멍 뚫린 항아리에 물 붓듯이 다른 이슈 계속 발생 -> Makefiile에서 target directory를 인식하지 못해서 발생하는 이슈

-> M = $(PWD) 라고 적혀 있는 곳이 상당히 많은데, 이 부분을 shell pwd로 수정 후 문제 해결

커널 일정 버전 이상부터는 shell pwd로 적어야 한다는 이야기 발견

2. ,ko 파일 까지 생성 완료 후 insmod시 invalid module format 에러 발생

가. 커널 버전과 모듈 버전이 달라서 생기는 이슈 -> uname -r 과 modinfo modname 을 입력해 보고 두 버전을 동일하게 맞춰주자.

나. 커널 버전과 모듈 버전 동일하나 문제 발생 : 이 경우에 속하였는데, 만약 커널 버전이 여러 개 깔려 있다면 커널 버전이 꼬여있을 수 도 있으므로 사용하는 커널만 남겨두고 나머지는 제거 후 재시도

<kernel 삭제>

---

uname -r -> 자신의 현재 kernel 버전 확인

dpkg –list | grep linux-image -> 제거할 kernel 확인

sudo apt-get purge <linux-image-version>

이후 무언가 안지워졌다는 로그는 찾아가서 rm -rf

---

sudo apt update && sudo apt upgrade

sudo apt remove –purge linux-headers-*

sudo apt autoremove && sudo apt autoclean

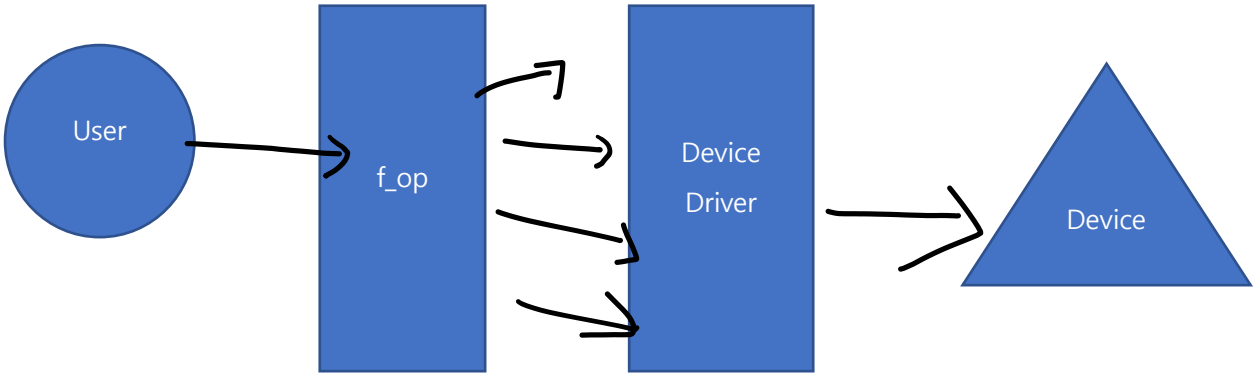
sudo apt install linux-headers-generic

이후 make 시 header가 없다는 에러가 발생할 수 있음.

sudo apt-get install linux-headers-$(uname -r)

해결이 안된다면, 문제가 발생한 지점에서 dmesg를 입력해 kernel message를 확인하자.

7. Device Driver



Device가 잘 작동할 수 있도록 device와 사용자 간의 interface 제공.

여러 device가 존재하므로 각 device를 구분하는 기준 필요 -> Major number (max = 4096)

Device driver를 구현 할 때, 모든 OS에서 구동되는 것이 중요함, 따라서 구현 할 때 하드웨어와 밀접한 부분, OS를 타는 부분을 구분지어서 구현하는 것이 정석

- ➔ mknod /dev/{name} c||b major # minor # 디바이스 드라이버 등록
- ➔ major #는 cat /proc/devices를 하면 할당되어 있는 major number를 확인 할 수 있음.

Device driver는 사용자 task가 file_operation 구조체에 정의되어 있는 함수를 통해 device file에 접근 할 때 호출 할 함수를 정의하고 구현해주는 것이 device driver이다. 이는 리눅스에서 device를 모두 파일로 인식하기 때문에 가능하다. 리눅스에서는 키보드, 마우스, 모니터 등 device들을 모두 파일로 인식하며, 이 파일들을 각각 다루기 위해 디바이스 드라이버 또한 나누어져 있다. 크게 3가지 종류로 분류하는데, character device driverr, block device driver, network device driver로 구분한다.

Character device driver와 block device driver는 page cahce를 사용하느냐 사용하지 않느냐로 구분 할 수 있다. Character device driver는 순차적으로 임의의 크기로 데이터 전송이 가능한 반면 block device driver는 정해진 크기에서 임의의 순서로 데이터 전송이 가능하다. 여기서 정해진 크기로 데이터 전송을 해야 하기 때문에, 사용자 100byte의 데이터만 요청을 하여도 1 page의 size를 읽어와서 거기서 100byte만 사용자에게 넘겨주는 식으로 처리를 해야한다. 같은 page를 매번 불러 읽어오려면 overhead가 발생하므로 cache concept을 사용하여 page cache를 적용한다. 또한, 차이점으로 character device driver는 read(), write()와 1:1 매칭되는 함수가 존재하지만. Block device drvier는 존재하지 않는다. 이유 또한 위의 이유와 동일한데, 데이터 통신이 큐와 page cahce를 사용하기 때문에 그에 맞는 인터페이스가 필요한 것이다.

이에 새로운 device driver를 구현하기 위해서는 4가지 단계가 기본적으로 요구된다.

1. Device driver 코어 함수를 구현하며, 이를 구현할 때에는 하드웨어 메뉴얼을 통해 구현한다.

2. 1)에서 작성한 함수를 리눅스에 등록하기 위해 wrapper 함수를 작성한다.

3. insmod를 이용하여 device driver를 커널에 등록한다.

4. Device driver를 호출하기 위한 진입점 (/dev/drivername)에 해당하는 장치 파일을 생성한다. (mknod or wrapper 함수에 내장 함수)

Goal : ZCU106 GPIO LED를 조작 할 수 있는 Device driver 구현

1. User <-> Device Driver

A. Write: write 0 -> LED OFF

Write 1 -> LED ON

-> Value를 어떤식으로 전달 할 것인가? User-level에서 scanf와 같은 방식이 가능할지?

Default: ./program argv1 argv2...

B. Read: Read current LED status : "Current LED status is ON / OFF"

-> user level terminal에서 message를 보려면 어떠한 method 사용하여야 하는지 조사 필요 -> 책에 있는 소스코드 받아쓰기 해보았을 때, 기본 printf는 보질 못하고 printk, prints?가 주로 보임.

2. Device Driver <-> LED

A. 함수 재 mapping -> f_op 구조체에 나와있는 함수를 구현해주면 될 듯 함. 기본적으로 필요한 함수 : LED_open, LED_release, LED_read, LED_write,

B. Connect between Device Driver and LED(Device) : 기존의 user – level에서는 mmap으로 연결 / kernel – level로 진입하면서 mmap불가. -> ioremap()이라는 함수

mmap() : user – level <-> Kernel level -> user – level에서 주소에 접근할 때 쓰이는 방법

ioremap() : Kernel – level <-> Kernel level (Safe) -> Kernel 내부에서 접근 할 때 쓰이는 방법

➔ ioremap이 system architecture를 탄다는 이슈를 접함. X86, ARM이든 잘 구동 되는지 확인 필요.

3. Petalinux & ZCU106 board에 올리는 방법

Step 1 : 안전한 방법으로 ZCU106보드에 GPIO LED FPGA를 올리고 시작

- ➔ **A. Petalinux SD 카드에 makefile과 소스코드 옮겨서 로컬에서 빌드 및 insmod 시도**
- ➔ 현재 결과 petalinux에 make build 파일이 존재하지 않음. (lib/modules/../build 존재X)
- ➔ Petalinux-config -c kernel로 이 또한 올릴 수 있다면 재 build 후 시도
- ➔ **B. Petalinux rebuild including user module**
- ➔ Google에 검색해본 결과 petalinux를 빌드 할 때 module을 포함해서 build할 수 있는 듯 함. 이 방법을 알아내면 build를 해서 local에서 작동여부 확인 가능.

Step 2: ZCU106보드에 LED FPGA 올리지 않고 petalinux만 올려서 시도.

- ➔ 이 step이 되지 않는다면 AXI BUS에 대해 조사해야 함. AXI가 GPIO LED를 mapping시켜 준다는 의미.