임베디드 펌웨어

< 펌웨어 >

1. 펌웨어

1) JTAG 하드웨어

▶ JTAG(Joint Test Action Group) : 임베디드 하드웨어 디버깅 인터페이스이며, 디지털 회로 테스트를 위한 산업 표준이다. JTAG 인터페이스는 IEEE 1149.1-1990 표준으로 정의되었으며, Boundary Scan라고도 불린다.

▶ Boundary Scan : 디지털 칩 내부의 논리 회로를 검사하는 방법을 제공한다. 이 기술을 사용하면 PCB(Printed Circuit Board)에서 실제 연결이 올바르게 이루어졌는지 확인하거나, 개별 칩의 핀 상태를 직접 읽고 쓸 수 있다.

▶ JTAG 인터페이스

: 일반적인 JTAG 인터페이스는 다음과 같은 4개의 신호 선으로 구성된다.

(1) TDI (Test Data In): JTAG 명령어와 데이터가 칩으로 들어가는 경로이다.

(2) TDO (Test Data Out): 칩에서 나오는 데이터 경로이다.

(3) TCK (Test Clock): JTAG 동작을 위한 클럭 신호이다.

(4) TMS (Test Mode Select): 현재 실행 중인 명령어나 상태를 변경하는데 사용된다.

▶ JTAG 활용

(1) 디버깅: JTAG 인터페이스를 통해 마이크로 프로세서나 마이크로 컨트롤러 등의 내부 상태를 직접 조사하거나 변경할 수 있다. 이 기능을 활용하여 하드웨어 레벨에서 소프트웨어 디버깅이 가능하다.

(2) Firmware Flashing: 일부 장치들은 JTAG 인터페이스를 통해 직접적으로 Firmware(내장 메모리)에 접근할 수 있다. 이런 경우, 손상된 Firmware 복구나 업그레이드 등에 활용할 수 있다.

하지만 모든 하드웨어가 완전한 JTAG 기능을 지원하지 않으며, 보안 문제 때문에 일부 기능이 제한되기도 한다(예: 모바일 장치).

▶ JTAG 도구

: 여러 종류의 공식 및 비공식 JTAG 도구가 있다. OpenOCD(Open On-Chip Debugger), UrJTAG 등은 오픈 소스 JTAG 디버깅 소프트웨어이다. 이 외에도 Segger J-Link, Xilinx Platform Cable, ARM Keil ULINK 등과 같은 상용 도구들도 많이 사용된다.

이런 도구들은 보통 특정 하드웨어 디버거(JTAG 어댑터)와 함께 사용되며, 이 디버거는 PC와 타겟 장치 사이에서 JTAG 신호를 변환하는 역할을 한다.

2) 스타트업 코드

▶ 펌웨어 스타트업 코드 : 임베디드 시스템이 부팅될 때 main()함수 이전에 가장 먼저 실행되는 코드이다. 이 코드는 하드웨어 초기화, 메모리 설정, 중요한 시스템 변수 및 레지스터 설정 등의 역할을 수행하며, 이 후에 메인 애플리케이션 코드를 호출한다.

▶ 펌웨어 스타트업 코드 수행 작업 순서

(1) 하드웨어 초기화: 필요한 경우, 스타트업 코드는 하드웨어를 초기화한다. 이는 특히 임베디드 시스템에서 중요한 단계이다. 예를 들어, 메모리 컨트롤러, 타이머, 입출력 장치 등을 설정한다.

(2) 스택 및 힙 초기화: 스타트업 코드는 프로그램의 스택과 힙 영역을 초기화한다. 이는 프로그램의 실행 동안 사용할 메모리를 준비하는 단계이다.

(3) 전역 변수 초기화: 프로그램의 전역 변수와 정적 변수를 초기화한다. 이 변수들은 프로그램의 수명 동안 계속 존재하므로, 프로그램이 시작될 때 초기값을 설정한다.

(4) 표준 라이브러리 초기화: 필요한 경우, 스타트업 코드는 표준 라이브러리를 초기화한다. 예를 들어, C++에서는 입출력 라이브러리, 예외 처리 메커니즘 등을 초기화한다.

(5) 메인 함수 호출: 마지막으로, 스타트업 코드는 프로그램의 메인 함수를 호출한다. 이는 프로그램의 실행을 시작하는 단계이다.

▶ 스타트업 코드 실행 순서

(1) Power-On Self-Test 수행(POST)

(2) 인터럽트 벡터 테이블 셋업

(3) 모든 인터럽트 실행 차단

(4) 장치 초기화(GPIO, SDRAM, Clock, Watch-dog 등)

(5) 메모리 영역 초기화(Stack, BSS 영역 등)

(6) 부트로더를 SDRAM으로 복사

(7) 부트로더 실행으로 점프

▶ 위와 같은 작업들은 대부분 어셈블리 언어나 저수준 C 언어로 작성되며, 특정 하드웨어 아키텍처와 칩셋에 따라 다르게 구현된다.

3) 메모리 초기화

▶ 펌웨어에서의 메모리 초기화는 시스템이 부팅될 때 수행되며, 이는 임베디드 시스템의 정상적인 동작을 위해 필수적인 과정이다. 이 초기화 과정은 주로 스타트업 코드에서 처리되며, 여러 단계를 거치게 된다.

▶ 메모리 초기화 과정

(1) 레지스터 설정: 프로세서의 레지스터들이 적절한 값으로 설정된다. 예를 들어, Stack Pointer(SP)는 RAM 영역 내에 적절한 위치(보통 RAM의 끝)를 가리키도록 설정된다.

(2) BSS(Block Started by Symbol) 세그먼트 초기화: BSS 세그먼트는 프로그램 내부의 전역 변수나 정적 변수 등을 저장하는 메모리 영역이다. 이 영역은 0으로 초기화되어야 한다.

(3) Data 세그먼트 초기화: Data 세그먼트는 초기값을 가진 전역 및 정적 변수들을 저장하는 메모리 영역이다. 프로그램이 로드될 때, 이 값들은 보통 플래시 메모리나 ROM에 저장된 값을 복사하여 초기화한다.

(4) Heap과 Stack 영역 설정: 동적 메모리 할당(heap)과 함수 호출(stack)에 사용되는 영역도 적절하게 설정되어야 한다.

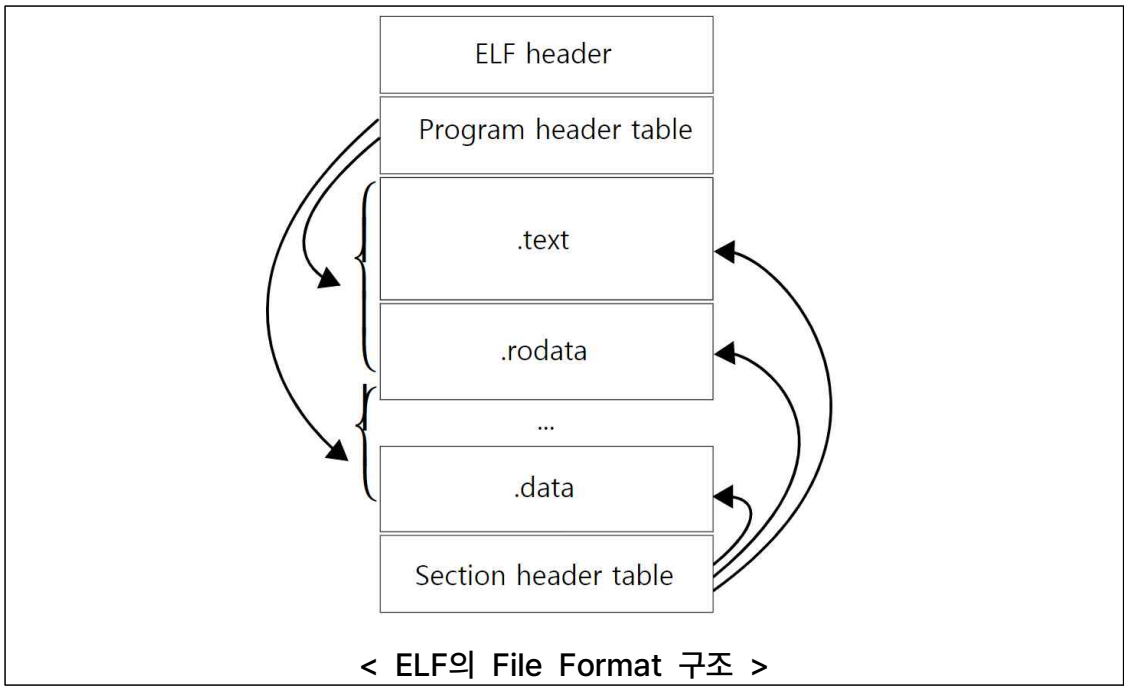
(5) 하드웨어 관련 메모리 맵 설정: 특정 하드웨어 컴포넌트들은 메모리 맵 IO를 사용하여 제어된다 (예: GPIO). 이런 컴포넌트들을 제대로 사용하기 위해서는 해당 주소가 적절하게 설정되어야 한다.

(6) MMU(Memory Management Unit) 설정 (만약 사용된다면): MMU는 가상 메모리 관리와 페이지 관련 작동 등 다양한 역할을 수행한다.

4) ELF 포맷과 binuils 도구

▶ ELF(Executable and Linkable Format): ELF파일은 실행 가능한 파일, 오브젝트 파일, 공유 라이브러리, 코어 덤프 등 다양한 형태의 이진 파일을 표현하는 표준 파일 포맷이다.(크로스 컴파일러 빌드 후 주로 사용) 이는 주로 UNIX 및 UNIX 계열 운영 체제에서 사용되며, 리눅스에서도 가장 일반적으로 사용되는 이진 파일 포맷이다.

▶ ELF 파일은 헤더와 여러 섹션 또는 세그먼트로 구성된다. 헤더는 파일의 전반적인 정보를 포함하며, 섹션 또는 세그먼트는 실제 코드, 데이터, 디버그 정보 등을 포함한다.



< ELF의 File Format 구조 >

▶ ELF Header

: ELF 파일의 가장 앞부분에 위치하며, 파일의 전체적인 정보(메타데이터)를 제공한다. 이 헤더는 파일의 종류(실행 가능 파일, 공유 객체 파일 등), 시스템 아키텍처(32비트, 64비트 등), 엔트리 포인트(프로그램의 시작 주소), 프로그램 헤더 테이블과 섹션 헤더 테이블의 위치 등을 포함하고 있다.

▶ Program Header Table

: ELF 파일 내에 있는 각 세그먼트(segment)의 위치와 크기, 메모리에 로드되는 위치, 접근 권한 등을 정의한다. 즉, 이 테이블은 실행 가능한 프로그램을 메모리에 로드하고 실행하는 데 필요한 정보를 제공한다. 일반적으로, 운영 체제는 프로그램 헤더 테이블에 따라 프로그램을 메모리에 로드하고 실행한다.

▶ Section Header Table

: ELF 파일 내에 있는 각 섹션(section)의 위치와 크기, 섹션의 이름, 섹션의 종류(코드, 데이터, 심볼 테이블 등) 등을 정의한다. 즉, 이 테이블은 컴파일러, 링커, 디버거 등 개발 도구가 ELF 파일을 처리하는 데 필요한 정보를 제공한다. 일반적으로, 운영 체제는 섹션 헤더 테이블을 사용하지 않지만, 개발 도구는 이 테이블을 통해 필요한 섹션을 찾아 처리한다.

▶ ELF의 자료형은 man elf에 존재한다.

자료형	자료형 설명
ElfN_Addr	- 프로그램 주소를 표현할 때 사용 - uintN_t 를 재정의
ElfN_Off	- 파일에서 Offset을 표현할 때 사용 - uintN_t 를 재정의
ElfN_Section	- section의 index를 표현할 때 사용 - uint16_t 를 재정의(16비트)
ElfN_Versym	- version symbol정보를 표시할 때 사용 - uint16_t 를 재정의(16비트)
ElfN_Byte	unsigned char를 재정의
ElfN_Half	- Half는 '하프 워드(half word)를 의미) (32비트의 반절) - uint16_t 를 재정의(16비트)
ElfN_Sword	int32_t 를 재정의(32비트)
ElfN_Word	uint32_t 를 재정의(32비트)
ElfN_Sxword	int64_t 를 재정의(64비트)
ElfN_Xword	uin64_t 를 재정의(64비트)
* ElfN은 Elf32 또는 Elf64로 표현된다.	

▶ binutils(GNU Binary Utilities)

: GNU 프로젝트에서 제공하는 이진 도구 모음이다. 이 모음에는 여러 가지 도구가 포함되어 있으며, 이러한 도구들은 이진 파일(특히 오브젝트 파일, 라이브러리, 실행 파일)을 생성, 수정, 분석하는 데 사용된다(컴파일과 디버깅에 관련된 여러 가지 tool을 제공한다).

▶ binutils 도구 모음

도구(명령어) 이름	설명
as	GNU 어셈블러로, 소스 코드를 오브젝트 파일로 변환한다.
ld	GNU 링커로, 여러 오브젝트 파일을 하나의 실행 파일로 연결한다.
ar	아카이브 파일(정적 라이브러리)을 생성하거나 수정하는 도구이다.
nm	오브젝트 파일이나 실행 파일에서 심볼을 나열한다.
objdump	오브젝트 파일이나 실행 파일의 정보를 보여주는 도구이다. 이는 바이너리 파일의 헤더 정보, 섹션 정보, 디스어셈블된 코드 등을 출력할 수 있다.
readelf	ELF 형식의 오브젝트 파일이나 실행 파일의 정보를 보여주는 도구이다. ELF 헤더, 섹션 헤더, 프로그램 헤더, 심볼 테이블 등의 정보를 출력한다.
strip	오브젝트 파일이나 실행 파일에서 디버그 정보나 심볼을 제거하는 도구이다.
addr2line	주어진 주소에 해당하는 파일 이름과 행 번호를 보여주는 도구이다. 주로 디버깅에 사용된다.
c++filt	C++ 및 Java 프로그램에서 사용되는 이름 맹글링(name mangling)을 디맹글링(demangling)하여 원래의 함수 이름을 복원해주는 도구이다.
gprof	프로파일링 데이터를 생성하고 분석하는 도구이다. 프로그램이 실행되는 동안 각 함수에서 얼마나 많은 시간이 소요되었는지 등의 정보를 제공한다.
size	오브젝트 파일이나 실행 파일의 섹션 사이즈를 보여주는 도구이다. 텍스트, 데이터, bss 섹션의 크기와 총합을 출력한다.
strings	오브젝트 파일이나 실행 파일에서 인쇄 가능한 문자열을 추출하는 도구이다.
objcopy	오브젝트 파일의 형식을 변환하거나, 오브젝트 파일을 수정하는 도구이다. 예를 들어, ELF 형식의 오브젝트 파일을 바이너리 형식으로 변환하는 데 사용할 수 있다.

2. 부트로더

1) 부트로더의 종류와 기능

▶ 부트로더는 컴퓨터가 부팅될 때 실행되며, 운영 체제를 메모리에 로드하고 실행

하는 역할을 하는 소프트웨어이다. 부트로더는 일반적으로 BIOS 또는 UEFI와 같은 펌웨어에 의해 호출되며, 하드 디스크, SSD, CD-ROM 등의 저장 매체에서 운영 체제 커널을 찾아 메모리에 로드한다. 보통 임베디드 시스템에서는 하드디스크와 같은 메모리가 없기 때문에 부트로더는 플래시 메모리에 저장되어 있고, 시스템이 부팅될 때, 부트로더는 보통 SDRAM이나 다른 형태의 RAM에 복사되어 실행된다.

▶ 부트로더의 주요 기능

(1) 하드웨어 초기화: 부트로더는 시스템 하드웨어를 초기화하고 적절한 상태로 설정한다. 이 과정에서 CPU, 메모리 컨트롤러, 그래픽 카드 등의 주요 하드웨어 구성 요소가 초기화된다.

(2) 운영 체제 커널 로딩: 부트로더는 저장 매체에서 운영 체제 커널을 찾아서 메모리에 로드한다.

(3) 커널 실행: 부트로더는 메모리에 로딩된 운영 체제 커널을 실행시킨다. 이 당시부터 운영 체제가 시스템 제어를 인수하여 사용자 모드 애플리케이션들이 동작할 수 있게 된다.

▶ 부트로더의 종류

(1) GRUB(Grand Unified Bootloader): 가장 널리 사용되는 멀티-부팅 부트로더 중 하나이다. GRUB은 여러 개의 운영 체제 중에서 선택하여 부팅할 수 있으며, 그래프형 사용자 인터페이스와 대화식 명령 줄 모드를 지원한다.

(2) LILO(Linux Loader): 리눅스 전용 부팅 로더이지만 현재는 대체적으로 GRUB에 의해 대체되었다.

(3) Syslinux: FAT 파일 시스템을 사용하는 편집 딱지 및 USB 드라이브 등과 같은 경량 매체에서 리눅스나 DOS를 부팅하기 위한 라이브러리이다.

(4) Das U-Boot(Universal Boot Loader): 주로 임베디드 시스템에서 사용되는 부트로더로, 다양한 아키텍처를 지원한다.

(5) Windows Boot Manager: Windows 운영 체제에 내장된 부트로더이다. 이 부트로더는 Windows Vista 이상의 버전에서 사용된다.

(6) EFI Boot Manager: UEFI(Unified Extensible Firmware Interface) 환경에서 사용되는 부트 매니저이다. 이것은 다양한 운영 체제를 지원하며, 펌웨어 수준에서 멀티-부팅 기능을 제공한다.

2) OS 부트과정

▶ OS 부팅과정

(1) 전원 켜짐(POWER ON): 사용자가 컴퓨터의 전원을 켜면, 하드웨어는 자체적인 초기화 과정을 시작한다. 이 과정을 Power-On Self-Test(POST)라 한다.

(2) BIOS 실행: POST가 완료되면, BIOS(Basic Input/Output System)가 실행된다. BIOS는 하드웨어를 초기화하고, 부트로더를 찾아 메모리에 로드하는 역할을 한다.
(3) 부트로더 실행: BIOS가 부트로더를 메모리에 로드하면, 부트로더는 운영 체제 커널을 디스크에서 찾아 메모리에 로드한다. 부트로더는 일반적으로 MBR(Master Boot Record) 또는 EFI(EFI System Partition)에 저장되어 있다.
(4) 커널 실행: 커널이 메모리에 로드되면, 커널은 시스템의 하드웨어와 소프트웨어를 초기화하고, 시스템 서비스와 데몬을 시작한다.
(5) 시스템 프로세스 시작: 커널이 필요한 서비스를 모두 시작하면, 시스템은 사용자 모드로 전환하고, 사용자가 로그인할 수 있도록 로그인 프롬프트를 표시한다. 이 단계에서, 사용자는 시스템을 사용할 수 있게 된다.
▶ OS 부팅과정 요약
: Power On → POST(Power On Self Test) → Boot Sector Loading → Boot Code Execute & Kernel Loder Loading → Kernel Loader Execute & Kernel Loading → Kernel Execute
▶ 부트로더의 OS 부팅 과정
(1) 부트로더 로딩: BIOS 또는 UEFI는 설정된 부트 순서에 따라 적절한 부트 장치(예: 하드 드라이브, SSD, CD-ROM 등)에서 부트로더를 찾아 메모리에 로드한다.
(2) 부트 설정 읽기: 일부 고급 부트로더(예: GRUB)는 사용자가 정의한 설정 파일을 읽는다. 이 설정 파일은 어떤 운영 체제를 기본적으로 부팅할지, 얼마나 오랫동안 사용자 입력을 기다릴지 등의 정보를 포함하고 있다.
(3) 운영 체제 선택: 만약 멀티-부팅 환경에서 실행되고 있다면, GRUB과 같은 고급 부트로더는 사용자가 어떤 운영 체제를 시작할지 선택하도록 할 수 있다.
(4) 운영 체제 커널 로딩: 선택된 (또는 기본적으로 지정된) 운영 체제의 커널 이미지를 저장 매체에서 찾아서 메모리에 로드한다.
(5) 커널 파라미터 전달: 필요한 경우, 부트로더는 커널 이미지와 함께 몇 가지 파라미터들(보통 "커맨드 라인"이라고 함)도 전달한다. 이 파라미터들은 커널 동작 방식에 영향을 줄 수 있다.
(6) 커널 실행: 마지막으로, 메모리에 로딩된 운영 체제 커널이 실행된다. 이 당시부터 운영 체제가 시스템 제어를 인수하여 사용자 모드 애플리케이션들이 동작할 수 있게 된다.

3) 플래시 메모리 관리

▶ 부트로더는 플래시 메모리를 관리하는 데 중요한 역할을 한다. 플래시 메모리는 주로 임베디드 시스템에서 운영 체제, 애플리케이션 코드, 데이터 등을 저장하는 데 사용되며, 부트로더는 이러한 정보를 올바르게 로드하고 관리해야 한다.

▶ 플래시 메모리(Flash Memory)

: 전기적으로 데이터를 지우고 다시 기록할 수 있는(electrically erased and reprogrammed) 비휘발성 컴퓨터 기억 장치를 말한다.

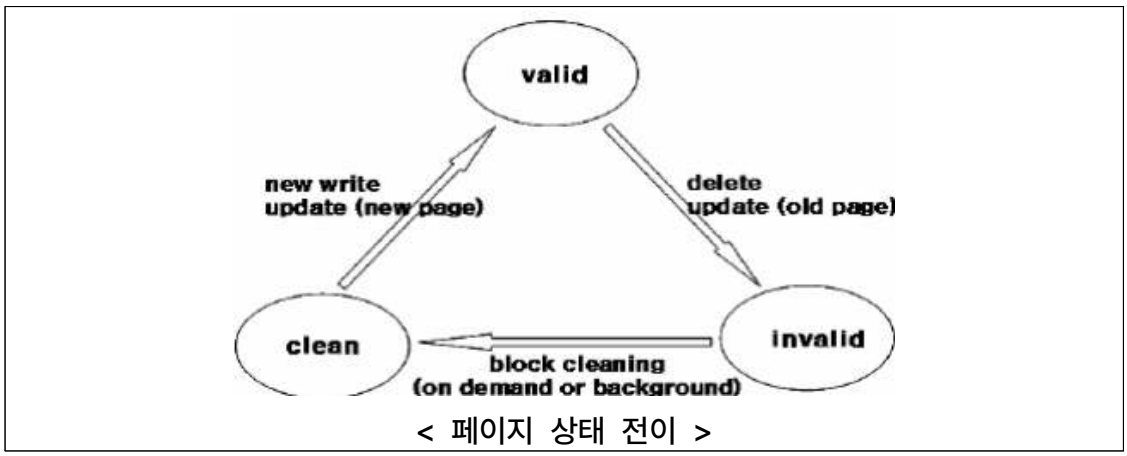
▶ 플래시 메모리의 특징

- 읽기, 쓰기, 삭제의 세 가지 연산 지원
- 연산 수행 단위의 비대칭성 (e.g. 읽기는 페이지 단위, 삭제는 블록 단위)
- 연산 수행 시간의 비대칭성 (e.g. 읽는데 20us, 쓰는데 200us, 삭제하는데 1.5ms)
- 덮어쓰기(Overwrite)가 불가능함
- 각 블록 당 제한된 숫자의 삭제 연산 횟수를 가지므로, 마모도 평준화(Wear leveling)를 고려해야 함

▶ 플래시 메모리에서 사용되는 용어

- 페이지: 메모리 연산 수행의 최소 단위
- 블록: 페이지의 집합
- 마모도 평준화(Wear leveling): 플래시 메모리의 각 블록은 제한된 삭제 연산 횟수를 가지므로 하나의 블록만을 쓰고 지우고 하다보면, 해당 블록을 사용할 수 없게 된다. 따라서 각 블록의 연산 횟수를 저장하여 이를 바탕으로 블록들을 재배치(remapping)하거나 분산(seperate) 하여 제품의 수명을 증대시키는 기법을 의미한다.

▶ 페이지 상태 전이



< 페이지 상태 전이 >

- clean: erase 연산을 수행하여 block 내의 모든 페이지가 지워졌을 경우, 해당 블록 내의 페이지를 clean 상태라 부른다.

- valid: clean 상태의 페이지에 데이터를 기록하면 해당 페이지는 valid 상태가 된다.
- invalid: valid 상태의 페이지의 데이터를 지우거나 갱신하여 더 이상 사용하지 않게 된 경우 이를 invalid 라 부른다.
- 그러므로 flash memory 에 유효한 페이지가 없더라도 clean 한 페이지가 존재하지 않을 수도 있다 (모든 페이지가 invalid 상태인 경우). 따라서 블록을 erase 하여 invalid 한 페이지를 clean 한 상태로 바꿔놓는 작업이 필요하다.

▶ 플래시 메모리 관리 방식

(1) 메모리 초기화: 부트로더가 실행되면 가장 먼저 특정 영역의 플래시 메모리를 초기화한다. 이 과정에서 필요한 경우 특정 하드웨어 설정을 수행하거나, 특정 영역의 메모리를 0으로 채우기도 한다.

(2) 운영 체제 로딩: 부트로더는 운영 체제 커널 이미지가 저장된 위치의 플래시 메모리에서 해당 이미지를 읽어들여 RAM에 로드한다.

(3) 애플리케이션 코드 및 데이터 로딩: 일부 시스템에서는 애플리케이션 코드나 필요한 데이터도 함께 로드해야 할 수 있다. 이 경우 부트로더는 해당 정보가 저장된 위치의 편성 메모리에서 이들을 읽어들여 적절한 위치에 저장한다.

(4) 편성 업데이트 처리: 일부 고급 부트로더(예: U-Boot)는 "in-place" 업데이트와 같은 기능을 제공하여 실행 중인 시스템의 소프트웨어를 업데이트할 수 있다. 이 경우 부트로더는 새롭게 업데이트된 이미지 파일을 받아서 적절한 위치에 있는 기존 이미지와 교체한다.

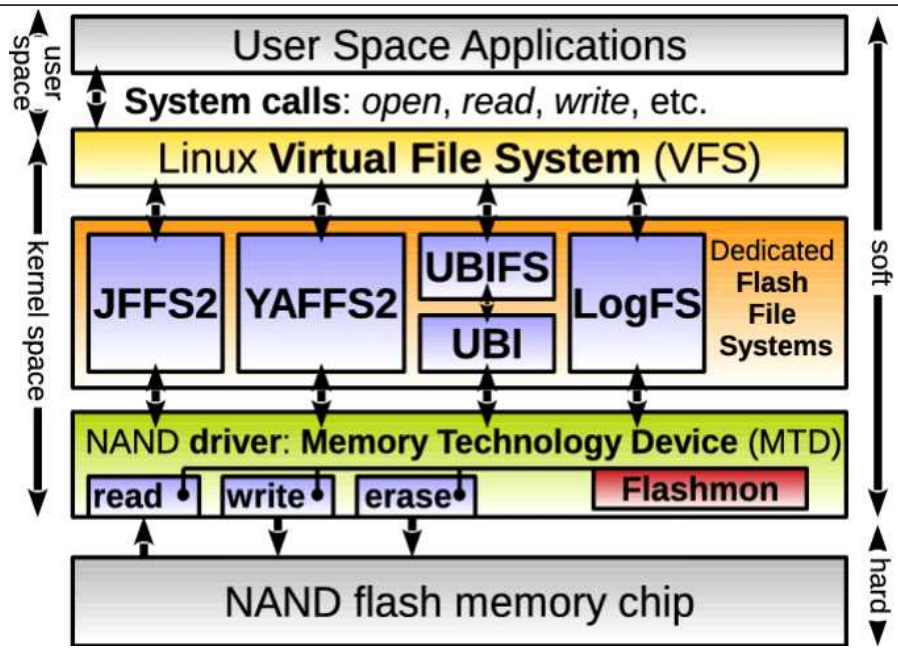
(5) 보안 기능 처리: 일부 보안 강화된 시스템에서는 부팅 과정 중에 서명 검증과 같은 보안 점검을 수행하기도 한다. 예를 들어, Secure Boot 기능이 활성화된 시스템은 UEFI 부트로더가 운영 체제 커널 이미지를 로드하기 전에 해당 이미지의 디지털 서명을 검증하여 신뢰할 수 있는 소스에서 제공된 이미지인지 확인한다.

▶ MTD(Memory Technology Device)

: MTD (Memory Technology Device)란 플래시 메모리와 통신하기 위한 리눅스의 장치 파일(device file) 이다. MTD는 메모리 장치의 물리적인 특성을 추상화하여, 플래시 메모리를 일반적인 블록 장치처럼 사용할 수 있게 해준다. 따라서 개발자는 플래시 메모리의 복잡한 특성을 직접 다루지 않고도, 플래시 메모리에 데이터를 읽거나 쓸 수 있다.

MTD의 주요 기능 중 하나는 플래시 메모리를 여러 개의 논리적인 파티션으로 나눌 수 있게 해주는 것이다. 이를 통해 하나의 플래시 메모리 장치를 여러 개의 독립적인 스토리지 장치처럼 사용할 수 있게 된다.

▶ MTD의 구조



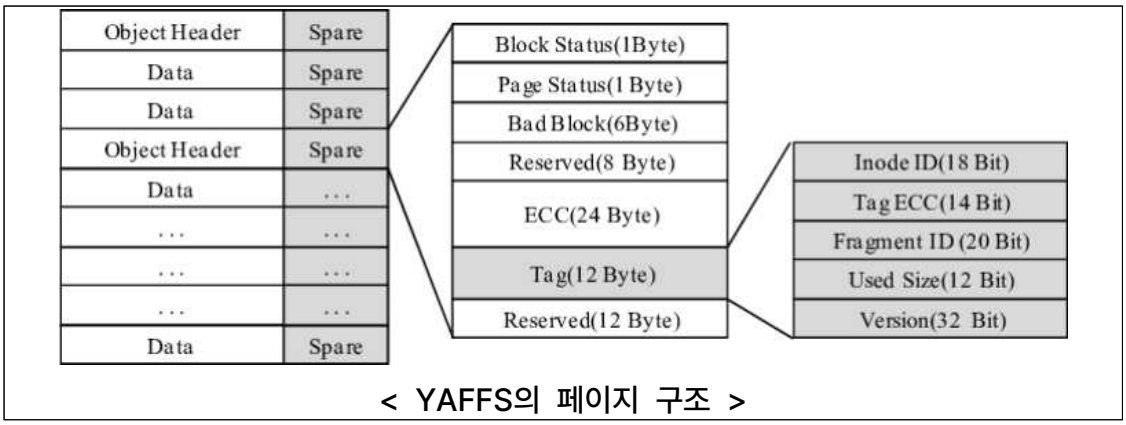
< MTD의 구조 >

- User Space Applications: System Call 을 호출하는 사용자 프로그램
- VFS (Virtual File System): 가상화된 파일 시스템
- Dedicated Flash File Systems: 플래시 메모리의 종류에 관계없이 MTD 가 제공하는 인터페이스를 이용하여 YAFFS, JFFS 등과 같은 플래시 전용 파일 시스템이나 FTL 등의 소프트웨어를 제공한다.
- MTD (Memory Technology Device): 실제 플래시 메모리에 명령을 내려 저수준 I/O 를 처리하며 MTD Glue Logic 에 일관된 인터페이스를 제공한다.
- NAND flash memory chip: 실제 물리적인 flash memory 장치

▶ YAFFS(Yet Another Flash File System)

: 플래시 메모리를 위해 특별히 설계된 파일 시스템이다. 특히 NAND 플래시 메모리에 대한 지원에 초점을 맞추고 있다. YAFFS는 2002년에 처음 발표되었으며, 주로 임베디드 시스템에서 사용된다.

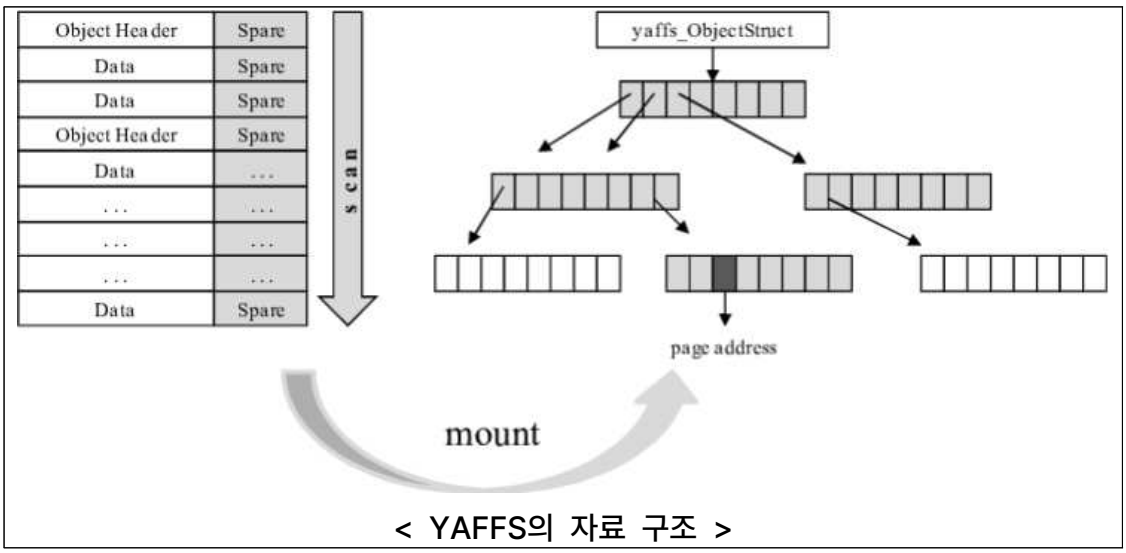
▶ YAFFS의 페이지 구조



< YAFFS의 페이지 구조 >

: YAFFS 는 NAND 플래시 메모리의 각 페이지를 'chunk' 라고 부르며, 각 페이지의 Spare 영역을 가지고 이를 tag 라 부른다. 각 chunk 는 header 혹은 data 이다. data 는 말 그대로 유저가 메모리에 기록한 데이터이고, header 는 파일에 대한 메타 데이터로 파일의 이름, 크기, 속성, 변경 시간 등의 정보가 저장되어진다.

▶ YAFFS의 자료구조



< YAFFS의 자료 구조 >

: YAFFS 는 mounting 시에 플래시 메모리에서 모든 헤더 chunk 들을 RAM 으로 읽어와서 파일 시스템 자료구조를 구축한다. 이를 통해 yaffs_ObjectStruct 를 읽어오고 읽어들인 yaffs_ObjectStruct 객체들을 서로 연결하여 파일 시스템의 트리 구조를 형성한다.

▶ JFFS(Journaling Flash File System)

: JFFS는 1999년에 처음 개발되었다. JFFS는 모든 쓰기 작업을 로그 형태로 플래시 메모리에 추가하므로, 파일 시스템의 상태를 간단한 방식으로 유지할 수 있다. 또한, JFFS는 플래시 메모리의 소거 사이클을 균등하게 분산시키는 데 도움이 된

다. 하지만, JFFS는 큰 용량의 플래시 메모리를 효율적으로 관리하는 데 어려움이 있다.

- 주요 특징

(1) 로그 구조: JFFS는 로그 구조 파일 시스템(Log-Structured File System)으로 설계되었다. 이는 모든 쓰기 작업이 순차적으로 수행되며, 삭제된 파일 또는 수정된 파일의 구역은 'dirty'로 표시되고, 나중에 가비지 컬렉션 과정에서 재사용된다.

(2) 플래시 메모리 최적화: JFFS는 플래시 메모리의 특성을 고려하여 설계되었다. 플래시 메모리는 한 번 쓰여진 블록을 덮어쓸 수 없기 때문에, JFFS는 새로운 데이터를 항상 빈 블록에 쓰고, 사용되지 않는 블록을 재사용할 수 있도록 관리한다.

(3) 장애 복구: JFFS는 시스템의 전원이 갑자기 차단되는 등의 예기치 않은 상황에서도 파일 시스템의 일관성을 유지하는 기능을 제공한다. 이는 특히 임베디드 시스템에서 중요한 기능이다.

▶ JFFS2(Journaling Flash File System version 2)

: JFFS2는 JFFS의 단점을 개선하기 위해 개발된 파일 시스템이다. JFFS2는 플래시 메모리의 사용을 최적화하기 위해 다양한 기능을 추가하였다. 예를 들어, JFFS2는 압축 기능을 제공하여 플래시 메모리의 사용 가능한 공간을 늘려준다. 또한, JFFS2는 플래시 메모리의 구조를 이해하고 이에 맞게 데이터를 쓰는 '플래시 공식 알고리즘'을 사용한다. 이를 통해 JFFS2는 플래시 메모리의 수명을 연장시키는 데 도움이 된다.

▶ LogFS

: 플래시 메모리를 위한 로그 구조 파일 시스템이다. 주로 대용량 NAND 플래시 메모리에 최적화되어 있으며, 임베디드 시스템뿐만 아니라 데스크탑이나 서버와 같은 일반적인 컴퓨팅 환경에서도 사용될 수 있다.

- 주요 특징

(1) 로그 구조: LogFS는 로그 구조 파일 시스템이다. 즉, 모든 쓰기 작업이 순차적으로 이루어지며, 수정이 필요한 경우 기존 데이터를 업데이트하는 대신 새로운 데이터를 파일 시스템의 끝에 추가한다. 이러한 방식은 플래시 메모리의 특성에 맞게 설계되었으며, 플래시 메모리의 수명을 연장하는 데 도움이 된다.

(2) 대용량 지원: LogFS는 대용량 플래시 메모리를 지원하도록 설계되었다. 일반적으로 로그 구조 파일 시스템은 작은 용량의 플래시 메모리에 적합하지만, LogFS는 특별한 알고리즘을 사용하여 큰 용량의 플래시 메모리를 효율적으로 관리할 수 있다.

(3) 장애 복구 기능: LogFS는 시스템의 전원이 갑자기 차단되는 등의 예기치 않

은 상황에서도 파일 시스템의 일관성을 유지하는 기능을 제공한다. 이는 특히 임베디드 시스템에서 중요한 기능이다.

(4) 플래시 메모리 최적화: LogFS는 플래시 메모리의 특성을 고려한 설계로, 플래시 메모리의 수명을 연장하고 성능을 최적화하는 데 도움이 된다.

4) 초기 RAM Disk 이미지

▶ 초기 RAM 디스크(Initial RAM Disk, initrd) 또는 초기 RAM 파일 시스템(Initial RAM Filesystem, initramfs)은 부팅 과정에서 임시적으로 사용되는 메모리 기반의 파일 시스템이다. 이들은 커널이 완전히 부팅되고 루트 파일 시스템이 마운트될 수 있도록 하는 중간 단계에서 필요한 모듈들을 로드하는 역할을 한다.

(1) initrd와 initramfs의 차이점: 초기에는 initrd가 널리 사용되었다. 이는 일종의 임시 블록 디바이스로, 커널에 의해 메모리로 로드된 후에 loopback 형태로 마운트된다. 반면, initramfs는 cpio (복사 파일 in/out) 아카이브를 통해 직접 메모리로 압축 해제되어, 복잡한 loopback 과정 없이 바로 접근 가능한 파일 시스템을 제공한다.

(2) 사용 목적: 운영 체제가 시작할 때 필요한 드라이버나 모듈들을 로드하기 위해 사용된다. 예를 들어, 루트 파일 시스템이 특정 RAID 배열 위에 위치하거나 특별한 네트워크 스토리지에 있는 경우 해당 하드웨어나 프로토콜을 지원하는 드라이버가 필요하며 이를 로드하기 위해 initrd나 initramfs가 사용된다.

(3) 작동 방식: 부팅 과정에서 부트로더 (예: GRUB)는 커널 이미지와 함께 선택적으로 초기 RAM 디스크 이미지를 메모리에 로드한다. 커널은 이 이미지를 사용하여 가상의 임시 파일 시스템을 생성하고 필요한 모듈들을 그곳에서 찾아서 로드한다.

(4) 생성과 관리: 대부분의 리눅스 배포판은 mkinitrd 나 mkinitramfs와 같은 도구를 제공하여 관련 모듈과 스크립트를 포함하는 적절한 초기 RAM 디스크 이미지를 생성하게 돕는다.

(5) 내부 구조: Initrd 나 Initramfs 내부에는 필요한 모듈, 설정 파일, 스크립트 등이 포함되어 있다. 가장 중요한 부분은 'init'라는 이름의 실행 파일로 이는 커널이 초기 RAM 디스크를 마운트 한 후에 가장 먼저 실행하는 프로그램이다.

(6) 사용 후 처리: Initrd 나 initramfs에서 필요한 모든 작업이 완료되면, 실제 루트 파일 시스템을 마운트하고 이로 전환한다. 그리고 나서 임시 파일 시스템은 해제(unmount)되고 메모리에서 해제(release)된다.

5) 네트워크 파일 시스템 이용

▶ 부트로더가 네트워크 파일 시스템을 이용하는 것은 주로 네트워크 부팅, 즉 원격 서버에서 호스트 컴퓨터를 부팅하는 상황에서 발생한다. 이러한 방식은 PXE(Preboot eXecution Environment)라는 표준을 사용하여 구현되며, TFTP(Trivial File Transfer Protocol)와 DHCP(Dynamic Host Configuration Protocol)를 사용하여 필요한 부팅 정보와 파일들을 전송한다.

(1) DHCP 요청과 응답: 컴퓨터가 시작되면, 네트워크 카드의 PXE 클라이언트는 DHCP 요청을 보낸다. 이 요청에는 PXE 클라이언트가 부팅에 필요한 정보(예: TFTP 서버의 IP 주소, 부팅할 파일의 이름 등)를 찾고 있다는 사실이 포함되어 있다. DHCP 서버는 이 요청에 응답하여 필요한 정보를 제공한다.

(2) TFTP를 통한 파일 전송: DHCP 응답을 받은 후, PXE 클라이언트는 TFTP 프로토콜을 사용하여 지정된 서버에서 필요한 파일들(예: 부트로더 이미지, 커널 이미지 등)을 다운로드한다.

(3) 부트로더 실행: 다운로드된 부트로더(예: GRUB, iPXE 등)가 실행된다. 이 때부터 부트로더가 제어권을 가진다.

(4) 커널 및 초기 RAM 디스크 로딩: 일반적으로 로컬 스토리지에서 수행하던 작업과 동일하게 진행되지만, 여기서도 TFTP나 HTTP 같은 프로토콜들을 통해 원격으로 커널 및 초기 RAM 디스크 이미지를 가져오게 된다.

(5) 루트 파일 시스템 마운팅: 네트워크 기반 루트 파일 시스템(NFS - Network File System), iSCSI(Internet Small Computer System Interface), AoE (ATA over Ethernet)와 같은 프로토콜들 중 하나를 사용해서 원격 루트 파일 시스템을 마운트 한다.

6) 부트로더 작성 및 타겟시스템 이식

▶ 부트로더를 작성하고 타겟 시스템에 이식하는 과정은 매우 복잡하며, 사용하는 하드웨어, 운영 체제, 프로그래밍 언어 등에 따라 크게 달라진다.

▶ 일반적인 절차

(1) 하드웨어 및 시스템 요구사항 파악: 부트로더를 작성하기 전에, 먼저 타겟 시스템의 하드웨어 구조와 운영 체제의 요구사항을 정확히 이해해야 한다. 예를 들어, CPU의 아키텍처(예: x86, ARM 등), 메모리 레이아웃, 입력/출력 인터페이스 등을 알아야 한다.

(2) 부트로더 코드 작성: 부트로더는 보통 어셈블리 언어와 C언어를 혼합하여 작성된다. 어셈블리 코드는 주로 초기 시작 단계에서 사용되며(CPU 초기화, 메모리 설정 등), C 코드는 상대적으로 복잡한 로직(커널 이미지 로딩 등)을 처리하는 데 사용된다.

(3) 컴파일 및 링크: 작성된 소스 코드는 특정 툴체인(크로스 컴파일러 등)을 사용하여 바이너리 형태로 컴파일되고 링크된다.
(4) 타겟 시스템에 설치: 생성된 부트로더 바이너리는 타겟 시스템의 특정 위치(예: 하드 디스크의 MBR 영역)에 설치되어야 한다.
(5) 시작 주소 설정 및 테스트: 일부 경우에서는 BIOS 또는 UEFI 설정에서 부팅 순서나 시작 주소를 변경해야 할 수도 있다. 모든 설치가 완료되면 실제 하드웨어나 에뮬레이터에서 부팅 과정을 테스트한다.
(6) 디버깅 및 최적화: 문제가 발생하면 디버거 도구를 사용하여 원인을 찾고 수정한다. 이 과정은 여러 번 반복될 수 있으며, 부트로더의 성능 최적화도 이 단계에서 수행된다.

3. 전원관리

1) 전원관리 하드웨어

▶ 전원 관리 하드웨어는 컴퓨터 시스템에서 전력 소비를 효율적으로 관리하는 데 중요한 역할을 한다. 이는 시스템의 전력 소비를 줄이고, 배터리 수명을 연장하며, 시스템의 성능과 안정성을 유지하는 데 필수적이다.

▶ 전원 관리 하드웨어 구성요소

(1) 전원 공급 장치(Power Supply Unit, PSU): PSU는 외부 전원 소스(예: 벽면 콘센트)로부터 에너지를 받아 컴퓨터 내부의 다양한 부품들이 사용할 수 있는 적절한 전압으로 변환한다. 일반적으로 PSU는 여러 개의 출력 볼트를 제공하여 CPU, 메모리, 디스크 드라이브 등 각 부품의 요구사항에 맞게 에너지를 공급한다.
(2) 배터리: 모바일 장치나 노트북 컴퓨터 등에서는 배터리가 주요 전원 공급 장치로 사용된다. 배터리 관리 시스템(Battery Management System, BMS)은 배터리 충전 상태, 온도 등을 모니터링하고 조절하여 최대 수명과 성능을 보장한다.
(3) 전압 레귤레이터(Voltage Regulator): 전압 레귤레이터는 입력 전압을 안정된 출력 전압으로 변환한다. 일반적으로 CPU와 같은 고성능 부품 근처에 위치해 있으며, 그들에게 필요한 정확한 작동 전압을 제공한다.
(4) 전력 관리 IC(Power Management ICs): PMICs는 배경 조명 제어부터 배터리 충전까지 다양한 기능들을 담당하며 여러 종류의 DC-DC 컨버팅 기능도 포함되어 있다.
(5) 시스템 관련 칩셋: 예를 들면 Southbridge와 같은 칩셋들은 종종 시스템 내에서 서로 다른 파워 스테이트 (예 : Suspend-to-RAM (S3),

Suspend-to-Disk (S4) 등)를 관리하는 역할을 한다.

(6) CPU: 현대의 CPU들은 자체적으로 전력 소비를 줄이기 위한 여러 가지 기능을 가지고 있다. 이는 주파수 스케일링, 동적 전압 조정 등 다양한 기능을 포함하고 있다.

2) OS 전원관리

▶ 운영 체제의 전원 관리 기능은 시스템의 전력 소비를 최적화하고, 배터리 수명을 연장하며, 전체적인 성능과 안정성을 유지하는 데 중요한 역할을 한다. 이는 하드웨어와 소프트웨어가 상호 작용하는 복잡한 과정을 포함한다.

▶ OS의 전원관리 방법

(1) 전력 상태(Power States): 대부분의 운영 체제는 여러 가지 전력 상태를 지원한다. 예를 들어, "작동 중(Active)" 상태에서는 모든 하드웨어 구성 요소가 정상적으로 작동하고, "대기(Sleep)" 또는 "절전(Suspend)" 상태에서는 비활성화할 수 있는 하드웨어 구성 요소를 비활성화하여 전력 소비를 줄인다. 또한 "종료(Shutdown)" 상태에서는 시스템의 모든 부분이 꺼진다.

(2) CPU 전력 관리: 운영 체제는 CPU 사용률에 따라 동적으로 CPU 클럭 속도와 전압을 조절할 수 있다(Dynamic Voltage and Frequency Scaling, DVFS). 이 기능은 일반적으로 CPU 주파수 스케일링이라고 부른다.

(3) 디바이스 관리: 운영 체제는 개별 장치들에 대해 다양한 절전 모드를 제공하며 필요에 따라 장치들을 활성화/비활성화 할 수 있다.

(4) 시스템 설정: 사용자가 시스템의 전원 설정(예: 화면 보호기 타임아웃, 자동 절전 모드 진입 등)을 조절할 수 있게 해준다.

(5) ACPI (Advanced Configuration and Power Interface): ACPI 표준은 운영 체제와 하드웨어 간의 inter-action을 정의하여 다양한 절전 기능과 함께 장치 구성 및 파워 관리 기능들이 언급된 표준이다.

(6) 에너지 효율 정책: 커널 스케줄러나 배경 작업 등도 에너지 효율에 영향을 준다. 예를 들어, 일부 작업을 배치하여 한 번에 처리하면 CPU가 더 긴 시간 동안 저전력 상태로 유지될 수 있다.

3) 부트로더의 전원관리

▶ 부트로더는 시스템이 부팅되는 초기 단계에서 동작하는 소프트웨어이다. 이 시점에서 전원 관리 기능은 대체로 제한적이며, 그 이유는 다음과 같다.

(1) 단순성 유지: 부트로더의 주요 목표는 운영 체제를 안전하게 시작하는 것이다.

복잡한 전원 관리 기능을 구현하면 부트로더 코드가 복잡해지고, 따라서 오류가 발생할 가능성이 높아진다.
(2) 하드웨어 접근 제한: 부트로더가 실행되는 시점에는 하드웨어 설정과 드라이버 로딩이 완전히 완료되지 않았을 수 있다. 따라서 전원 관리 기능에 필요한 하드웨어 접근 권한이 제한적일 수 있다.
(3) 시간 제약: 부트로더의 실행 시간은 상대적으로 짧으므로, 복잡한 전원 관리 기능을 적용하기에 충분하지 않을 수 있다.

< OS 포팅 >

1. 리눅스 내부구조 개요 및 포팅

1) 커널의 소스 트리 구조

▶ 커널 소스 트리는 운영체제의 커널을 구성하는 소스 코드들이 계층적으로 정리되어 있는 구조를 의미한다.

(1) /arch: 리눅스 커널 기능 중 하드웨어 종속적인 부분들이 구현된 디렉토리이다. 이 디렉토리는 각각의 하드웨어 아키텍처에 대한 코드 및 종속적인 커널 코드를 포함한다. 예를 들어, x86, arm 등의 하드웨어 아키텍처에 대한 코드가 여기에 위치한다. 문맥 교환과 같은 하드웨어 종속적인 태스크 관리 부분은 arch/x86/kernel 디렉토리에 구현되어 있다. arch/x86/boot 디렉토리에는 시스템의 초기화 때 사용하는 부트스트랩코드가 구현되어 있다. arch/x86/mm에는 메모리 관리자 중에서 페이지 부재 결함처리 같은 하드웨어 종속적인 부분이 구현되어 있다. arch/x86/lib에는 커널이 사용하는 라이브러리 함수가 구현되어 있다.

(2) /include: 이 디렉토리는 커널, 라이브러리 함수, 드라이버들에서 사용되는 헤더 파일들을 포함하고 있다.

(3) /drivers: 여기에는 다양한 장치 드라이버들의 코드가 있다.

(4) /fs: 파일 시스템과 시스템 호출 관련 코드가 저장된 디렉터리이다.

(5) /net: 네트워크 프로토콜 스택과 관련된 모든 것을 담당하는 디렉터리이다.

(6) /kernel: 이 디렉터리는 실제적인 커널(운영체제의 핵심 부분)의 기능 위한 코드를 담고 있다. (ex: 스케줄링 알고리즘, IPC 등).

(7) /init: 시스템 시작 초기화 과정에서 필요한 함수와 데이터가 저장된 위치이다. 커널의 초기화 부분, 즉 커널의 메인 시작 함수가 구현되어 있다.

(8) /scripts: 컴파일과 같은 빌드 과정에서 사용되는 스크립트 파일들이 있는 곳이다.

(9) /mm : 메모리 관리 파일들이 있는 곳이다.

(10) /lib : 라이브러리 함수들이 있는 곳이다.

(11) /tools : 유틸 도구들이 있는 곳이다.

(12) /ipc: 리눅스 커널이 지원하는 프로세스간 통신 기능이 구현된 디렉토리이다.

2) 커널 빌드 과정 개요

▶ 리눅스 커널 빌드

: 리눅스 운영체제의 핵심인 커널을 사용자의 필요에 맞게 설정하고, 그 설정에 따라 소스 코드를 컴파일하여 실행 가능한 바이너리 형태의 커널 이미지를 생성하는 과정을 말한다.

▶ 리눅스 커널 빌드가 요구되는 환경

(1) 새로운 하드웨어 지원: 새로운 하드웨어나 플랫폼을 지원하기 위해 특정 드라이버나 모듈을 포함시키거나 제외시키는 경우

(2) 최적화: 시스템 성능을 최적화하기 위해 필요하지 않은 기능들을 제외하거나, 특정 기능들을 활성화하는 경우

(3) 보안 강화: 보안 패치를 적용하거나, 보안 관련된 특별한 기능들을 활성화하는 경우

(4) 신규 기능 개발 및 테스트: 새로운 기능 개발이나 버그 수정 등 개발/연구 목적으로 커널 코드를 직접 수정하고 그 결과를 확인하기 위한 경우

▶ 리눅스 커널 빌드 과정

(1) 커널 소스 코드 다운로드: 커널 소스 코드를 직접 다운로드 받는다. 이는 주로 리눅스 커널 아카이브(https://www.kernel.org/)에서 진행할 수 있다.

(2) 커널 설정(config): .config 파일을 생성하거나 수정하여 빌드할 커널의 옵션을 설정한다. 이 과정에서는 make menuconfig, make xconfig, make oldconfig 등의 명령어를 사용할 수 있다.

(3) 컴파일(compilation): 설정이 완료되면, 실제로 소스 코드를 컴파일하는 작업을 한다. 일반적으로 make 명령어를 사용하여 이 작업을 수행한다.

(4) 모듈 설치(module installation): 필요한 모듈들을 설치한다. 리눅스 커널은 여러 가지 기능들을 모듈 형태로 제공하며, 이들 중 필요한 것들만 선택하여 설치할 수 있다.

(5) 커널 설치(kernel installation): 컴파일된 새로운 커널 이미지와 관련 파일들을 적절한 위치에 복사하고, 부트로더에 새롭게 빌드된 커널 정보를 업데이트 한다.

(6) 재부팅(rebooting): 마지막으로 시스템을 재부팅하여 새롭게 빌드 및 설치된 커널로 부팅한다.

3) 커널 구성(configuration) 방법

▶ 리눅스 커널 구성 방법

(1) 커널 소스 코드 다운로드: 리눅스 커널의 공식 웹사이트인 kernel.org에서 원하는 버전의 커널 소스 코드를 다운로드 받는다.

(2) 커널 설정: make menuconfig 명령어를 사용하여 커널을 구성한다. 이 명령어는 텍스트 기반의 메뉴 인터페이스를 제공하며, 여기에서 필요한 모듈과 옵션들을 선택할 수 있다.

(3) 컴파일: make 명령어를 사용하여 실제로 커널을 빌드(컴파일)한다. 이 과정에서는 앞서 설정된 옵션에 따라서만 필요한 부분들이 컴파일된다.
(4) 모듈 설치: make modules_install 명령어를 사용하여 필요한 모듈들을 시스템에 설치한다.
(5) 커널 설치: make install 명령어를 실행하여 새롭게 빌드된 커널 이미지와 관련 파일들을 /boot 디렉터리 등 적절한 위치에 복사하고, 부트 로더(grub 등)가 새롭게 빌드된 커널 이미지를 인식할 수 있도록 설정 파일도 업데이트 한다.
(6) 재부팅: 마지막으로 시스템을 재부팅하여 새롭게 빌드 및 설치된 리눅스 커널로 부팅한다.

2. 리눅스 부팅

1) 리눅스 부팅 과정

▶ 리눅스 시스템의 부팅 과정은 여러 단계를 거쳐 진행되며, 각 단계에서는 하드웨어 초기화, 커널 로딩, 시스템 초기화 등 다양한 작업이 이루어진다.
(1) BIOS (Basic Input/Output System) 단계: 컴퓨터가 전원을 켜면 가장 먼저 BIOS가 실행된다. BIOS는 하드웨어를 검사하고 초기화하는 POST(Power-On Self Test)를 수행한다. 그런 다음 설정된 부트 순서에 따라 부트로더가 있는 장치(일반적으로 하드 디스크)를 찾아 그 안의 부트로더에 제어권을 넘긴다.
(2) 부트로더 (Bootloader) 단계: 리눅스에서는 GRUB(Grand Unified Bootloader)나 LILO(Linux Loader) 같은 부트로더가 사용된다. 부트로더는 커널 이미지와 initrd 이미지(필요한 경우)를 메모리에 로드한다.
(3) 커널 시작: 메모리에 로딩된 커널이 실행되고, 필요한 초기화 작업을 수행한다. 이 때 하드웨어 장치들의 드라이버도 로드된다.
(4) init 프로세스 시작: 커널이 모든 초기화 작업을 마치면 init 프로세스 (PID 1인 프로세스)를 시작한다. 리눅스 배포판에 따라 System V init, Upstart, systemd 등 다양한 init 시스템 중 하나가 사용될 수 있다.
(5) 런레벨(Runlevel)/타겟(Target): init 시스템은 설정된 런레벨 혹은 타겟(시스템마다 표현이 조금씩 다름)에 따라 필요한 서비스와 데몬(운영체제 백그라운드에서 실행되는 프로세스)들을 시작한다.
(6) 로그인 프롬프트 나타남(등장): 모든 서비싱 및 데몬들이 성공적으로 시작되면 사용자는 로그인 프롬프트를 볼 수 있게 되며, 이제 사용자 계정 정보를 입력하여 시스템을 사용할 수 있게 된다.

2) init 스크립트

▶ 리눅스 시스템에서 init 스크립트는 시스템 부팅 과정에서 실행되는 일련의 스크립트이다. 이들은 커널이 로드된 후에 실행되며, 시스템의 런레벨(runlevel) 또는 타겟(target)에 따라 다양한 서비스와 데몬을 시작하거나 중지한다.

(1) System V init: 전통적인 리눅스 시스템에서 사용하는 init 시스템이다. /etc/inittab 파일을 통해 초기화 과정을 제어하며, /etc/rc.d 아래에 있는 각각의 런레벨 별로 서비스를 시작하거나 종료하는 스크립트가 있다.

(2) Upstart: Ubuntu에서 개발된 모던한 init 시스템으로, 이벤트 기반으로 작동하여 부팅 속도를 개선하고 동시성을 처리하는 것을 목적으로 한다. Upstart는 /etc/init 디렉터리 아래에 있는 .conf 파일들로 구성된다.

(3) systemd: 최신 리눅스 배포판에서 가장 널리 사용되는 init 시스템이다. systemd는 /lib/systemd/system/와 /etc/systemd/system/ 디렉터리 아래에 있는 .service, .socket, .target, .mount, 등등 다양한 유닛 파일들로 구성된다.

▶ 각 init 스크립트 혹은 유닛 파일은 보통 '서비스'라고 하는 백그라운드 프로세서(데몬)를 시작하거나 중지하는 역할을 한다. 이러한 스크립트 혹은 유닛 파일은 서비스가 어떻게, 언제 시작되어야 하는지, 어떻게 종료되어야 하는지 등의 정보를 담고 있다.

3) busybox와 셸

▶ Busy Box : 리눅스 시스템에서 사용되는 다양한 표준 유닉스 유틸리티를 하나의 실행 파일로 결합한 소프트웨어이다. 이는 임베디드 시스템, 부팅 가능한 CD, 간단한 시스템 복구 등과 같은 상황에서 매유 유용하며, 최소한의 디스크 공간만을 필요로 한다.

▶ BusyBox에 포함된 명령어들은 ls, cp, mv, cd 등 기본적인 파일 조작 명령어부터 grep, awk, sed와 같은 텍스트 처리 도구까지 다양하다. 그 외에도 네트워크 관련 명령어(ping, ifconfig 등), 프로세스 관리 명령어(ps, top 등) 그리고 init 스크립트 실행을 위한 init 도구도 포함되어 있다.

▶ 쉘(Shell) : 사용자와 운영체제 사이의 인터페이스를 제공하는 프로그램이다. 쉘은 사용자가 입력하는 커맨드를 해석하고 해당 커맨드를 실행한다. 또한 스크립팅 기능을 제공하여 일련의 커맨드들을 자동으로 실행할 수 있게 해준다.

▶ 리눅스에서 가장 널리 사용되는 쉘에는 bash(Bourne Again Shell), sh(Bourne SHell), csh(C SHell), ksh(Korn SHell), zsh(Z SHell) 등이 있다. 각 쉘은 기본적인 목적인 '명령 해석 및 실행'에 대해 공통점을 가지지만 그 외의 세부적인 기능이나 문법 면에서 차이점을 보입니다. 예를 들면 bash와 zsh는 고급 프로그래밍 기능과 사용자 친화적인 환경(명령 자동완성 등)을 제공하지만 sh는 최소화된 환경 아래서 동작하기 위해 설계되었습니다.

4) 커널 모듈 관리

▶ 리눅스 커널 모듈은 커널의 기능을 확장하는 방법 중 하나이다. 이는 디바이스 드라이버, 파일 시스템, 네트워크 프로토콜 등과 같은 기능을 제공하며, 필요에 따라 동적으로 로드하거나 언로드할 수 있다.

▶ 모듈 관리 명령어

(1) lsmod: 현재 로드된 모듈의 목록을 보여준다.

(2) insmod: 모듈을 커널에 삽입(로드)한다. 이 때, 의존성 있는 다른 모듈들이 먼저 로드되어 있어야 한다.

(3) rmmod: 특정 모듈을 언로드한다. 이 때 해당 모듈을 사용하는 프로그램이 없어야 한다.

(4) modprobe: insmod와 rmmod보다 더 고급 기능을 제공한다. modprobe는 의존성 있는 다른 모듈들도 함께 로드하거나 언로드할 수 있다.

(5) depmod: 의존성 정보를 생성하여 /lib/modules/$(uname –r)/ modules.dep 파일에 저장한다.

▶ 모듈 파일

: 리눅스에서 각 커널 모듈은 .ko(Kernel Object) 확장자를 가진 파일로 존재하며, 대부분 /lib/modules/$(uname -r)/ 디렉터리 아래에 위치해 있다.

▶ 모듈 파라미터

: 몇몇의 커널 모듈들은 작동 방식을 변경하기 위한 파라미터를 받아들일 수 있다. 이러한 파라미터는 insmod 혹은 modprobe 명령 실행 시 옵션으로 지정할 수 있다.

올바르게 작동하는 시스템에서는 대부분의 경우 자동으로 필요한 드라이버와 프로토콜 등의 커널 모듈이 로딩되지만, 특별한 하드웨어나 설정 등으로 인해 직접적인 관리가 필요할 수도 있다.

5) 공유 라이브러리 관리

▶ 리눅스에서 라이브러리는 주로 공유된 코드 조각을 포함하는 파일이다. 이들은 여러 프로그램에서 공통적으로 사용되는 기능을 제공하며, 메모리 사용량을 줄이고 코드 재사용성을 높여준다. 라이브러리 관리는 이런 라이브러리 파일들의 설치, 업데이트, 검색 등의 작업을 포함한다.

▶ 라이브러리 종류

(1) 정적 라이브러리(Static Libraries): .a 확장자를 가지며, 컴파일 시에 바로 실행 파일에 링크된다. 이렇게 되면 실행 파일은 해당 라이브러리 없이도 동작할 수 있지만, 크기가 커진다.

(2) 공유 라이브러리(Shared Libraries): .so 확장자를 가지며, 프로그램 실행 시

동적으로 로드되고 링크된다. 이렇게 되면 여러 프로그램들이 하나의 공유 라이브러리 인스턴스를 공유할 수 있어 메모리 효율성과 디스크 공간 효율성 측면에서 유리하다.

▶ 라이브러리 관련 명령어

(1) ldconfig: 공유 라이브러리 캐시(/etc/ld.so.cache)를 생성하거나 재구성하는 명령어이다.

(2) ldd: 특정 실행 파일에서 의존하는 공유 라이브러리 목록을 출력하는 명령어이다.

(3) nm: 오브젝트 파일(.o), 아카이빙된 오브젝트(.a), 공유 오브젝트(.so) 내부의 심볼(symbol) 정보를 확인하는 명령어이다.

▶ 라이브러리 탐색 경로

: 공유 라이브러리 탐색 경로는 주로 /etc/ld.so.conf와 그 하위 디렉터리 내부의 설정파일들에 의해 결정된다(include /etc/ld.so.conf.d/*.conf). 추가적으로 환경변수 LD_LIBRARY_PATH도 사용할 수 있다.

하지만 일반적으로 패키지 매니저(apt, yum, dnf 등)를 통해 설치된 라이브러리는 적절한 위치(/usr/lib, /usr/local/lib 등)에 설치되므로, 직접 경로를 설정할 필요는 드물다.

< 디바이스 드라이버 개발 >

1. 디바이스 드라이버 개념

1) 디바이스 드라이버의 개념

▶ 디바이스 드라이버는 컴퓨터의 운영 체제와 하드웨어 장치 사이에서 인터페이스 역할을 하는 소프트웨어이다. 디바이스 드라이버는 운영 체제가 하드웨어를 제어하고, 하드웨어에서 발생하는 이벤트를 해석하는 데 필요한 기능을 제공한다.

▶ 디바이스 드라이버의 중요한 개념

(1) 하드웨어 추상화: 디바이스 드라이버는 특정 하드웨어 장치의 세부사항을 숨기고, 일관된 인터페이스를 제공하여 프로그램들이 직접적으로 하드웨어를 조작하지 않고도 해당 장치를 사용할 수 있게 한다.

(2) 커널 모듈: 대부분의 디바이스 드라이버는 커널 모듈로 구현된다. 이러한 형태로 구현되면 시스템 부팅 후에도 로딩과 언로딩을 할 수 있으며, 전체 커널을 다시 컴파일하지 않고도 새로운 장치 지원을 추가하거나 기존 지원을 삭제할 수 있다.

(3) 인터럽트 처리: 많은 하드웨어 장치들은 작업 완료나 오류 상황 등 특정 이벤트 발생 시 인터럽트를 발생시킨다. 따라서 디바이스 드라이버는 해당 인터럽트를 처리하는 코드도 포함해야 한다.

(4) I/O 작업: 디바이스 드라이버는 사용자 레벨 애플리케이션과 하드웨어 사이에서 데이터 전송 역할도 한다. 이때 주로 입출력(I/O) 연산 방식(직접 I/O, 메모리 맵핑 I/O 등)과 버퍼링 방식(블록 버퍼링, 순환 버퍼링 등) 등 여러 가지 기법들을 활용한다.

(5) 전원 관리: 최신의 디바이스 드라이버들은 종종 전원 관리 기능을 포함한다. 이는 장치의 전력 사용을 최적화하고, 필요 없는 경우 장치를 절전 모드로 전환하는 등의 작업을 포함한다.

2) 디바이스 드라이버의 종류

(1) 그래픽 드라이버: 그래픽 카드나 온보드 그래픽 처리 장치를 제어한다. 이들은 화면 출력을 관리하고, 고급 3D 렌더링 기능을 제공하는 등의 역할을 한다.

(2) 네트워크 드라이버: 이더넷 카드, Wi-Fi 어댑터, 블루투스 모듈 등 네트워크 관련 장치를 제어한다.

(3) 사운드 드라이버: 사운드 카드나 온보드 오디오 처리 장치를 제어하여 오디오 출력 및 입력을 관리한다.

(4) 입력 장치 드라이버: 키보드, 마우스, 터치패드, 조이스틱 등의 입력 장치를 제어한다.

(5) 저장장치 드라이버: 하드 디스크, SSD, USB 메모리 스틱 등의 저장장치를 제어한다.

(6) 프린터 드라이버: 프린터와 같은 출력 장비를 제어하여 문서 인쇄 기능을 지원한다.

(7) USB 호스트 컨트롤러 드라이버: USB 포트와 연결된 다양한 종류의 USB 장비들을 감지하고 통신하는 데 필요한 기본적인 인터페이스를 제공한다.

(8) 카메라/웹캠 드라이버 : 내장 혹은 외부 웹캠 및 카메라와 같은 이미징 디바이스들에 대해 작동하게 해준다.

3) 리눅스 커널 모듈

▶ 리눅스 커널 모듈은 커널의 기능을 확장하는 방법 중 하나이다. 이는 디바이스 드라이버, 파일 시스템, 네트워크 프로토콜 등과 같은 기능을 제공하며, 필요에 따라 동적으로 로드하거나 언로드할 수 있다.

▶ 모듈 관리 명령어

(1) lsmod: 현재 로드된 모듈의 목록을 보여준다.

(2) insmod: 모듈을 커널에 삽입(로드)한다. 이 때, 의존성 있는 다른 모듈들이 먼저 로드되어 있어야 한다.

(3) rmmod: 특정 모듈을 언로드한다. 이 때 해당 모듈을 사용하는 프로그램이 없어야 한다.

(4) modprobe: insmod와 rmmod보다 더 고급 기능을 제공한다. modprobe는 의존성 있는 다른 모듈들도 함께 로드하거나 언로드할 수 있다.

(5) depmod: 의존성 정보를 생성하여 /lib/modules/$(uname –r)/modules.dep 파일에 저장한다.

(6) register_chrdev(): 이 함수는 문자 디바이스 드라이버를 커널에 등록하는 역할을 한다. 이 함수를 호출하면, 커널은 디바이스 번호를 할당하고, 해당 디바이스 드라이버의 파일 연산 함수들에 대한 포인터를 유지한다. 이 함수는 성공 시 0 이상의 값을 반환하고, 실패 시 음수 값을 반환한다.

(7) unregister_chrdev(): 커널에 등록된 문자 디바이스 드라이버를 등록 해제한다.

(8) request_region(): 이 함수는 I/O 포트 영역을 할당받는 역할을 한다. 이 함수를 호출하면, 커널은 지정된 I/O 포트 영역이 사용 가능한지 확인하고, 사용 가능하다면 해당 영역을 디바이스 드라이버에 할당한다. 이 함수는 성공 시 할당된 I/O 포트 영역에 대한 포인터를 반환하고, 실패 시 NULL을 반환한다.

(9) release_region(): I/O 포트 영역을 반환하는 역할을 한다.
(10) module_init(): 리눅스 커널 모듈 프로그래밍에서 사용되는 매크로로, 모듈이 커널에 로드될 때 호출되는 초기화 함수를 지정하는데 사용된다. 이 매크로는 인자로 초기화 함수의 이름을 받는다. 이 인자에 하나 이상의 함수를 넣어 사용할 수 있다.
(11) module_exit(): 모듈이 커널에서 제거될 때 호출되는 종료 함수를 지정하는데 사용된다.
▶ 모듈 파일
: 리눅스에서 각 커널 모듈은 .ko(Kernel Object) 확장자를 가진 파일로 존재하며, 대부분 /lib/modules/$(uname -r)/ 디렉터리 아래에 위치해 있다.
▶ 모듈 파라미터
: 몇몇의 커널 모듈들은 작동 방식을 변경하기 위한 파라미터를 받아들일 수 있다. 이러한 파라미터는 insmod 혹은 modprobe 명령 실행 시 옵션으로 지정할 수 있다.
올바르게 작동하는 시스템에서는 대부분의 경우 자동으로 필요한 드라이버와 프로토콜 등의 커널 모듈이 로딩되지만, 특별한 하드웨어나 설정 등으로 인해 직접적인 관리가 필요할 수도 있다.

2. 디바이스 드라이버

1) 표준 문자드라이버 API

▶ 리눅스에서 문자 디바이스 드라이버는 주로 일련의 바이트를 읽고 쓰는 인터페이스를 제공하는 장치를 제어하는 데 사용된다. 이러한 장치에는 시리얼 포트, 병렬 포트, 사운드 카드 등이 포함될 수 있다.
문자 디바이스 드라이버는 표준 API를 통해 운영 체제와 상호 작용하며, 이 API는 주로 다음과 같은 함수들을 포함한다.
(1) open(): 디바이스 파일을 열 때 호출되며, 드라이버 초기화와 관련된 작업을 수행한다.
(2) close(): 디바이스 파일을 닫을 때 호출되며, 필요한 정리 작업을 수행한다.
(3) read(): 사용자 공간 버퍼로 데이터를 읽어올 때 호출된다.
(4) write(): 사용자 공간 버퍼에서 데이터를 쓸 때 호출된다.
(4) ioctl() (입출력 제어): 특정 장치에 대한 다양한 커맨드나 요청을 처리하는 데 사용된다.
(5) mmap() : 메모리 맵핑 입출력 연산에 대해 구현할 수 있다.

2) 시스템 콜에 의한 드라이버

▶ 시스템 콜(system call)은 사용자 모드에서 실행 중인 프로그램이 커널 모드에서 제공하는 서비스를 요청할 수 있는 인터페이스이다. 이러한 서비스에는 파일 열기, 네트워크 소켓 생성, 메모리 할당 등이 포함된다.

▶ 디바이스 드라이버와 시스템 콜의 관계

(1) 시스템 콜을 통한 디바이스 접근: 프로그램은 open(), read(), write(), close() 등의 시스템 콜을 사용하여 디바이스 드라이버와 상호 작용한다. 예를 들어, 파일을 읽으려면 우선 open() 시스템 콜로 파일을 열고, 그 다음 read() 시스템 콜로 데이터를 읽는다. 이때 실제로 디바이스에서 데이터를 읽는 작업은 해당 디바이스의 드라이버가 담당한다.

(2) 시그널과 인터럽트 처리: 디바이스 드라이버는 하드웨어 인터럽트를 처리하고 필요에 따라 사용자 공간 프로그램에 시그널을 보내는 역할도 한다.

(3) ioctl() 시스템 콜: ioctl()은 "입출력 제어"를 두루마리하며, 표준 입출력 함수들로는 수행하기 어려운 다양한 종류의 장치 제어 작업을 수행하는 데 사용된다.

3) 커널 모듈 원격 디버깅

▶ 커널 모듈의 원격 디버깅은 개발자가 원격 위치에 있는 시스템에서 실행 중인 커널 모듈을 디버깅할 수 있게 해주는 과정이다. 이를 위해 가장 일반적으로 사용되는 도구는 gdb(GNU Debugger)와 kgdb(Kernel GNU Debugger)이다.

원격 디버깅을 설정하려면 두 대의 컴퓨터가 필요하다. 하나는 '목표(target)' 시스템이며, 여기서 디버깅하려는 커널 모듈이 실행된다. 다른 하나는 '호스트(host)' 시스템으로, 여기서 gdb나 kgdb를 실행하여 목표 시스템을 제어한다.

▶ 원격 디버깅 절차

(1) 디버그 지원으로 커널 빌드: 목표 시스템에서 실행할 커널은 디버그 지원을 포함하여 빌드되어야 한다. 이를 위해 .config 파일에서 관련 옵션(예: CONFIG_DEBUG_INFO, CONFIG_GDB_SCRIPTS 등)을 활성화해야 한다.

(2) 목표 시스템 준비: 목표 시스템에서 kgdboc(kernel gdb I/O module)를 로드하고, 통신 방식(예: tty, USB, Ethernet 등)과 포트를 설정한다.

(3) 호스트 시스템 준비: 호스트 시스템에서 gdb를 시작하고, 목표 커널의 vmlinux 파일(디버그 정보가 포함된 것)을 로드한다.

(4) 디버거 연결: 호스트와 목표 사이에 연결을 설정한다(gdb 내부의 target remote 명령 사용).

(5) 디버깅 시작: 이제 호스트 시스템에서 gdb inter-action을 통해 원격 디버깅 작업(브레이크 포인트 설정, 변수 검사 등)을 수월하게 진행할 수 있다.

3. 디바이스 드라이버와 커널 서비스

1) 커널의 주요자료 구조

▶ 리눅스 커널은 다양한 자료 구조를 사용하여 시스템의 상태와 동작을 관리한다.

▶ 커널의 주요자료 구조

(1) 프로세스 디스크립터 (task_struct): 이 자료 구조는 실행 중인 각 프로세스에 대한 정보를 담고 있다. 프로세스 ID, 상태, 우선순위, 부모 및 자식 프로세스 ID, 사용중인 메모리 주소 공간 등의 정보가 포함된다.

(2) 파일 디스크립터 (file_struct): 열려 있는 각 파일에 대한 정보를 저장하는 데 사용된다. 파일의 현재 위치, 접근 권한, 참조 카운트 등이 포함된다.

(3) Inode (inode_struct): 파일 시스템에서 각 파일과 디렉토리에 대한 메타데이터를 저장하는 데 사용되는 중요한 자료 구조이다. Inode 번호, 파일 타입(일반 파일, 디렉토리 등), 소유자 및 그룹 ID, 접근 권한 및 시간 스탬프 등이 포함된다.

(4) 소켓 디스크립터 (socket_struct): 네트워크 통신을 위해 생성된 각 소켓에 대한 정보를 저장한다.

(5) Superblock: Superblock은 리눅스 파일 시스템에서 사용되는 메타데이터 구조이다. Superblock은 해당 파일 시스템의 전반적인 정보를 포함하고 있으며, 예를 들어 블록 크기, Inode 수, 마운트 상태 등을 기록한다.

(6) Dentry (Directory Entry): Dentry는 디렉터리 엔트리와 관련된 정보를 담고 있는 자료 구조이다. Dentry는 디렉터리 이름과 해당 이름에 맵핑된 Inode 번호 등을 유지한다.

(7) 소켓 버퍼 (socket buffer): 소켓 통신에서 데이터 전송 및 수신을 위해 사용되는 버퍼입니다. 소켓 버퍼에 데이터가 채워지면 네트워크 인터페이스로 전송되거나 응용 프로그램으로 전달된다.

(8) 메모리 페이지(Page): 리눅스 커널은 가상 주소 공간과 물리적 메모리 사이의 매핑을 관리하기 위해 페이지 단위로 작업한다.

(9) 페이지 디렉터(Page Directory)와 페이지 테이블(Page Table): 가상 주소 공간의 페이지들과 물리적 메모리 페이지들 간의 매핑 정보를 유지하는 자료구조이다.

(10) 페이지 캐시(Cache): 최근 접근된 페이지들을 보관하여 I/O 성능 개선을 위

해 사용된다.

2) 디바이스 드라이버에서의 버퍼관리

▶ 버퍼는 장치와 시스템 간의 데이터 전송을 보조하며, 특히 장치의 작동 속도가 메인 메모리나 CPU의 속도보다 느릴 때 유용하다.

버퍼링(buffering)은 데이터를 일시적으로 저장하는 공간인 버퍼를 사용하는 것이다. 이것은 입출력 연산의 효율성을 높이고, 사용자 공간과 커널 공간 사이, 그리고 디바이스 드라이버와 하드웨어 장치 사이에서 데이터를 안전하게 전송하는 데 도움을 준다.

▶ 디바이스 드라이버 버퍼 종류

(1) 커널 버퍼(Kernel Buffer): 커널 공간에 위치한 이러한 버퍼는 디바이스 드라이버가 직접 제어하며, 하드웨어 장치로부터 데이터를 읽거나 하드웨어 장치에 데이터를 쓸 때 중간 저장소 역할을 한다.

(2) 사용자 버퍼(User Buffer): 이것은 사용자 공간에 위치하며 애플리케이션에 의해 제어된다. 디바이스 드라이버는 보안상의 이유로 직접 접근할 수 없으므로, copy_to_user() 및 copy_from_user()와 같은 함수들을 통해 안전하게 데이터를 전송한다.

3) 커널 메모리 할당과 해제

(1) kmalloc()와 kfree(): 이들은 커널 공간에서 메모리를 할당하고 해제하는 가장 기본적인 함수이다. kmalloc()은 특정 크기의 연속적인 물리 메모리를 할당하며, kfree()는 kmalloc()에 의해 할당된 메모리를 해제한다.

(2) vmalloc()와 vfree(): vmalloc()은 비연속적인 물리 메모리를 연속적인 가상 주소 공간으로 매핑하여 할당한다. 이는 큰 블록의 메모리가 필요할 때 유용하며, 결과적으로 페이지 단위로 분할된 여러 개의 물리 페이지로 구성된다. vfree()는 vmalloc()에 의해 할당된 메모리를 해제한다.

(3) get_free_page(), free_page(), alloc_pages(), __get_free_pages(), __free_pages() 등: 이들 함수는 페이지 단위로 직접적으로 동작하여 커널에서 사용할 수 있는 특정 크기(일반적으로 4KB)의 페이지 프레임을 직접 요청하거나 반환한다.

(4) kmem_cache_create(), kmem_cache_alloc(), kmem_cache_free(), kmem_cache_destroy(): 이들 함수는 SLAB 알로케이터라고 부르는 고급 캐시 시스템을 사용하여 작동한다. SLAB 알로케이터는 자주 사용되거나 특정 타입의 객체에 대한 재사용 가능한 캐시를 생성하고 관리함으로써 성능을 개선한다.

(5) dma_alloc_coherent(), dma_free_coherent(): DMA(Direct Memory Access) 버퍼용으로 사용되며, DMA가 가능한 연속된 물리 주소 공간을 할당 및 해제하는데 사용된다.

4) 상호배제 지원함수

▶ 상호배제(Mutual Exclusion)는 여러 프로세스나 스레드가 공유 자원에 동시에 접근하는 것을 방지하는 메커니즘이다. 이를 통해 데이터 경쟁 조건(race condition)이 발생하는 것을 막을 수 있다.

▶ 리눅스 커널 상호배제 기법

(1) 세마포어(Semaphore): 세마포어는 가장 기본적인 동기화 메커니즘이며, 한 번에 하나 이상의 프로세스나 스레드가 임계 영역(critical section)에 진입할 수 있도록 한다. 리눅스 커널에서는 sema_init(), down(), up() 등의 함수를 통해 세마포어를 제공한다.

(2) 뮤텍스(Mutex): 뮤텍스는 한 번에 오직 하나의 스레드만이 임계 영역에 진입할 수 있도록 하는 상호배제 메커니즘이며, 대부분의 경우 세마포어보다 더 간단하고 빠르다. 리눅스 커널에서는 mutex_init(), mutex_lock(), mutex_unlock() 등의 함수를 통해 뮤텍스를 제공한다.

(3) 스핀락(Spinlock): 스피닝은 잠긴(lock된) 락을 계속해서 폴링하면서 대기하는 것이다. 즉, CPU 시간을 소비하면서 "회전"한다. 이 방식은 공유 자원에 대한 접근 시간이 매우 짧은 경우 유용하며, 리눅스 커널에서는 spin_lock(), spin_unlock() 등의 함수를 사용하여 구현된다.

(4) 리더-라이터 락(Read-Write Lock): 이러한 종류의 락은 데이터 구조체가 여러 개의 읽기 연산과 적은 수의 쓰기 연산으로 구성된 경우 유용하다. 여러 개의 읽기 연산이 동시에 일어날 수 있지만, 쓰기 연산이 일어날 때는 해당 작업이 완료될 때까지 다른 모든 읽기/쓰기 연산을 차단한다. 리눅스 커널에서는 read_lock(), read_unlock(), write_lock(), write_unlock() 등의 함수를 통해 이를 제공합니다.

5) 동기/비동기 드라이버 개념

▶ 드라이버는 하드웨어와 운영 체제 사이에서 중재하는 역할을 하는 소프트웨어 컴포넌트이다. 드라이버는 동기식 방식과 비동기식 방식으로 구현될 수 있다.

(1) 동기 드라이버: 동기 드라이버에서 요청은 순차적으로 처리된다. 즉, 한 번에 하나의 요청만 처리하며, 해당 요청이 완료될 때까지 기다린다. 이런 방식은 프로그래밍하기가 상대적으로 간단하지만, I/O 작업을 기다리는 동안 CPU가 블로킹

(blocking)되므로 효율성이 떨어질 수 있다. 예를 들어, 파일 시스템에 대한 읽기 요청을 생각해보면, 동기적인 방식에서는 읽기 작업이 완료되고 데이터가 사용 가능해질 때까지 프로그램 실행이 일시 중지된다.

(2) 비동기 드라이버: 비동기 드라이버에서는 여러 요청을 동시에 처리할 수 있다. 즉, 한 번에 여러 개의 I/O 작업을 시작하고 완료 알림을 받으면 해당 작업을 종료한다. 이렇게 하면 I/O를 기다리는 동안 다른 작업을 계속할 수 있으므로 CPU 사용률과 시스템 성능을 향상시킬 수 있다. 비동기적인 방식에서도 파일 읽기를 예로 들면, 읽기 작업은 백그라운드에서 진행되며 프로그램은 다른 일들(다른 I/O 연산 포함)을 계속 처리할 수 있다. 데이터가 준비되면 적절한 콜백 함수나 인터럽트를 통해 알림 받게 된다.

6) 스케줄러를 이용한 대기

▶ 스케줄러는 운영 체제의 중요한 부분으로, 프로세스와 스레드가 CPU를 공유하는 방식을 결정한다. 이를 통해 시스템의 전반적인 성능과 응답 시간을 최적화한다.

▶ 대기(Waiting) 상태는 프로세스가 특정 이벤트(예: I/O 작업 완료, 세마포어 또는 락 획득 등)가 발생할 때까지 CPU를 사용하지 않고 대기하는 상태이다. 대기 상태의 프로세스는 스케줄러에 의해 실행되지 않는다.

▶ 리눅스 커널에서는 schedule() 함수를 호출하여 명시적으로 CPU에서 현재 실행 중인 프로세스나 스레드를 제거하고 다른 프로세스나 스레드에게 제어권을 넘길 수 있다. 이 함수 호출은 일반적으로 I/O 요청이 시작되거나, 락을 기다리거나, 타이머가 만료되는 등의 조건에서 발생한다.

▶ schedule() 함수 호출 후에는 리눅스 커널 스케줄러가 실행 가능한 다른 태스크 중 하나를 선택하여 실행한다. 대기 중인 태스크(즉, I/O 작업이 완료되거나 락이 해제된 경우)가 다시 실행 가능한 상태가 되면, 해당 태스크는 다시 실행 큐에 추가된다.

7) 커널 타이머

▶ 커널 타이머는 운영 체제의 중요한 기능 중 하나로, 특정 시간 간격 후에 실행되어야 하는 작업을 스케줄링하는 데 사용된다. 이를 통해 다양한 시간 관련 작업을 처리할 수 있다. 예를 들어, 타임아웃을 구현하거나 정확한 시간 지연을 생성하거나, 주기적으로 반복되는 작업(예: 상태 확인 또는 업데이트)을 스케줄링하는 등의 기능을 수행한다.

(1) 고수준 타이머 인터페이스: timer_setup(), mod_timer(), del_timer() 등의 함수를 사용하여 커널 내에서 재사용 가능한 일반적인 타이머를 설정하고 관리할

수 있다. 이러한 타이머들은 소프트웨어에 의해 특정 시점에 실행되도록 스케줄링된 콜백 함수와 연결된다.

(2) 저수준 타이머 인터페이스 (hrtimers): 고분해능(high-resolution) 태스크와 같은 더욱 정밀한 요구 사항에 대응하기 위해 hrtimers가 도입되었다. hrtimers API(hrtimer_init(), hrtimer_start(), hrtimer_cancel() 등)는 나노초 단위의 정밀도로 작동한다.

8) 세마포어

▶ 세마포어는 다중 프로세싱 환경에서 동기화 문제를 해결하는 데 사용되는 중요한 도구이다. 이는 공유 자원에 대한 동시 접근을 제어하고, 임계 영역에 대한 상호 배제를 보장하는 역할을 한다.

▶ 세마포어는 정수 값과 함께 작동하는 변수로 볼 수 있으며, 기본적으로 두 가지 주요 연산, 즉 wait (또는 P 연산)과 signal (또는 V 연산)을 지원한다.

(1) Wait (P) 연산: 세마포어의 값이 0보다 크면 값을 감소시키고, 그렇지 않으면 프로세스를 대기 상태로 만든다. 이것은 일반적으로 임계 영역에 진입하기 전에 수행된다.

(2) Signal (V) 연산: 세마포어의 값을 증가시키고, 만약 어떤 프로세스가 대기 중이라면 하나를 깨운다. 이것은 일반적으로 임계 영역에서 나온 후에 수행된다.

9) 인터럽트 서비스

▶ 인터럽트는 컴퓨터 시스템에서 중요한 이벤트가 발생했음을 CPU에 알리는 메커니즘이다. 이러한 이벤트는 하드웨어 장치에서의 입력(예: 키보드 키 누름, 네트워크 패킷 도착 등)이거나, 예외 조건(예: 나눗셈 오류, 페이지 폴트 등)일 수 있다.

▶ 인터럽트가 발생하면 현재 실행 중인 프로세스나 스레드를 일시 중지하고 해당 인터럽트를 처리하는 인터럽트 서비스 루틴(ISR, Interrupt Service Routine)이라고 하는 코드로 제어를 전환한다. ISR은 인터럽트 요청을 처리한 후에 원래의 작업으로 제어를 반환한다.

▶ 인터럽트 서비스 루틴

(1) 단기간 실행: ISR은 가능한 한 빠르게 실행되고 종료되어야 한다. 그렇지 않으면 시스템의 성능에 부정적인 영향을 줄 수 있다.

(2) 재진입 가능해야 함: 동일한 인터럽트 요청이 다시 발생할 수 있으므로 ISR은 재진입 가능해야 한다. 즉, 여러 번 호출될 때 안전하게 동작해야 한다.

(3) 원자성: ISR 내에서 수행되는 연산들은 원자적으로 수행되어야 한다. 즉, 한 번 시작된 연산은 완료될 때까지 다른 인터럽트에 의해 방해받지 않아야 한다.

(4) 상태 저장 및 복원: ISR이 호출되면 현재 CPU 상태(레지스터 값 등)를 저장하고, ISR 종료 후에 이 상태를 복원하여 원래의 작업을 계속할 수 있도록 해야 한다.

10) DMA(DIrect Memory Access) 개념

▶ 리눅스 커널에서는 DMA(Direct Memory Access) 방식으로 입출력 연산을 처리하기 위해 DMA 버퍼도 활용한다. DMA 방식은 CPU 없이도 메모리와 I/O 디바이스 간에 직접적으로 데이터 전송할 수 있게 해줌으로써 성능 향상에 기여한다.