PART II

커널의 시작

start_kernel은 어떻게 호출될까?

커널의 실제 시작 함수는 start_kernel() 함수다. 이 함수는 다시 100여 개의 함수들을 호출하면서 부팅을 진행한다. 하지만 startk_kernel() 함수가 호출되기 전에 커널 컴파일을 통해 얻어진 zImage의 압축 해제, 페이지 디렉터리 구성과 같은 기초적인 작업이 먼저 수행되어야 한다. 바로 이번 파트에서 start_kernel() 함수가 불리기 전까지 일련의 과정을 알아볼 것이다.

이번 파트에서 다루는 모든 코드는 ARM 어셈블리로 작성되어 있다. 부트로더로부터 PC(Program Counter)가 start 레이블로 이동되면서 기초 작업이 수행된다.

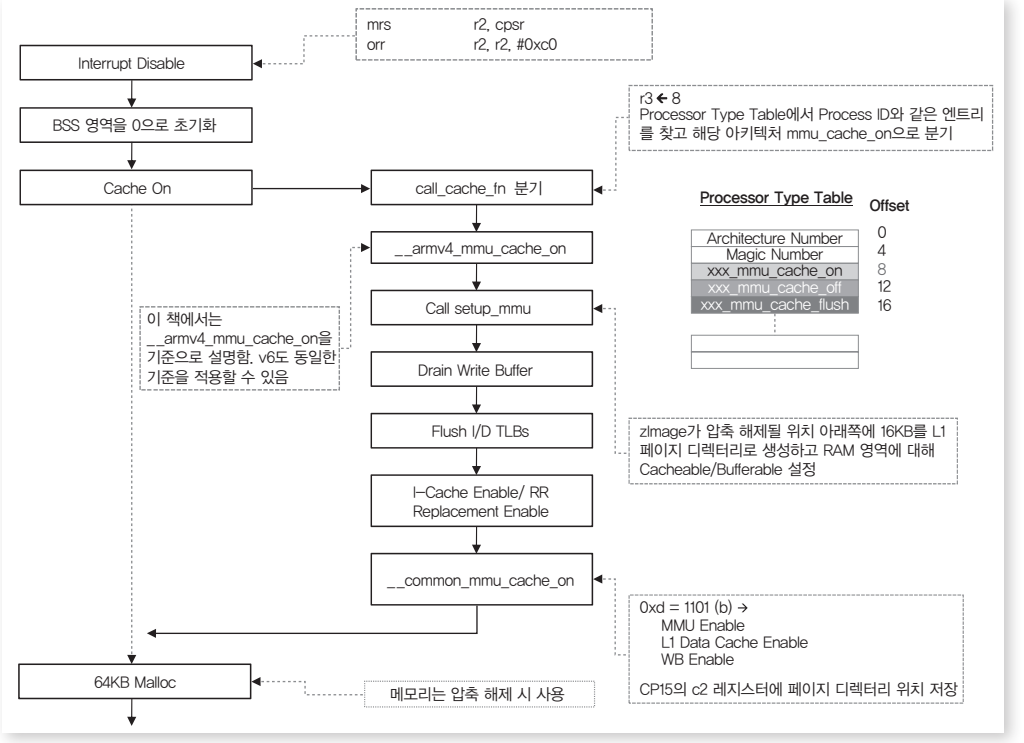
start_kenel이 호출되는 단계는 크게 3단계로 구분할 수 있는데, 1단계는 커널 이미지인 zImage 압축을 해제하기 위한 준비 단계로 프로세서 버전에 맞는 프로세서 타입 테이블로부터 캐시 켜기, 끄기, 플러시 등을 수행하고, MMU를 위한 16KB 페이지 디렉터리를 구성한다. 2단계는 zImage의 압축을 해제한다. 마지막 3단계에서는 프로세서 및 머신 정보와 부트로더로부터 넘겨온 atag 정보의 유효성 등을 검사한 후 MMU를 활성화하는 단계를 거쳐 커널의 시작인 start_kernel() 함수를 호출하게 된다.

그림 PART II–1은 이번 파트에서 알아볼 커널의 주요 내부 흐름(flow)을 보여준다.

CHAPTER 5

커널 압축 해제 준비하기

이번 장은 압축된 커널 이미지(zImage)를 해제하기 위해 필요한 단계들에 대해서 알아볼 것이다. 압축 해제 준비 단계로는 인터럽트 비활성화, 동적 메모리 할당, BSS 영역 초기화, 페이지 디렉터리 초기화, 캐시 켜기 등을 수행한다. 압축을 해제하기 위한 준비 중 가장 중점을 두는 부분은 zImage의 압축이 해제되는 위치를 기준으로 하위 16KB에 페이지 디렉터리를 위한 공간을 구성하고, CP15의 c2 레지스터에 페이지 디렉터리의 위치를 저장하는 것이다. ARM에서 페이지 디렉터리는 4GB의 메모리를 1MB 섹션으로 관리한다. 따라서 4GB를 관리하기 위해서는 1MB 단위의 4,096개 엔트리가 필요하고 각 엔트리는 32비트 워드로 관리되기 때문에 총 16KB(4Byte × 4,096엔트리 = 16KB)가 필요하게 된다. 또한 페이지 디렉터리의 위치에 해당하는 엔트리에 cacheable과 bufferable을 설정하여 페이지 디렉터리가 캐싱되어 빠르게 접근될 수 있도록 한다. 그림 5-1은 이번 장에서 알아볼 커널 코드의 내부 흐름을 도식화한 것이다.



:: 그림 5-1 start 레이블에서 압축 해제 준비까지 흐름도

5.1 부트로더에 이어 첫 스타트 끊기 – start 레이블

부트로더에 의해서 하드웨어 및 소프트웨어에 대한 기본적인 초기화가 이루어진 이후에 가장 먼저 실행되는 것은 arch/arm/boot/compressed/head.S에 있는 start 레이블에 있는 코드들이다.[12] start 레이블에서는 부트로더로부터 아키텍처 ID와 atags(7.4절 참고) 정보를 전달받는다.

또한 인터럽트 비활성화 및 레지스터를 초기화하고 not_relocated 레이블로 점프한다. 코드 5-1을 자세히 살펴보자.

:: 코드 5-1 arch/arm/boot/compressed/head.S의 start 레이블

```
start:
    .type   start,#function
    ...
1:  mov r7, r1    @ 아키텍처 ID 저장                                    ❶
    mov r8, r2    @ atags 포인터 저장
    ...
#ifndef __ARM_ARCH_2__                                                  ❷
    ...




    mrs r2, cpsr                  @ CPSR 가져오기
    orr r2, r2, #0xc0             @ 6, 7번 비트에 IRQ, FIQ 끄도록 설정
    msr cpsr_c, r2                @ CPSR에 반영해서 인터럽트 끄기
#else
    teqp    pc, #0x0c000003       @ 인터럽트 끄기                       ❸
#endif
    ...

.text                                                                   ❹

    adr  r0, LC0
    ldmia    r0, {r1, r2, r3, r4, r5, r6, ip, sp}
    subs    r0, r0, r1

beq not_relocated
```

start 레이블의 코드 섹션 ❶에서는 부트로더로부터 전달된 아키텍처 ID와 atags 정보를 레지스터 r7, r8에 각각 저장한다.

12) 부트로더는 다음과 같이 5가지 기능을 제공해야 한다. 1) RAM 초기화, 2) 시리얼포트 초기화, 3) 머신 타입 찾기, 4) 커널 tagged list 구성, 5) 커널 이미지로 제어 이관

코드 섹션 ❷에서는 인터럽트를 비활성화한다. IRQ와 FIQ를 모두 비활성화시키며, 인터럽트를 비활성화시키는 방법은 ARM 프로세서의 버전에 따라서 구현이 다르다. __ARM_ARCH_2__는 ARM2, ARM3 프로세서에 대해서 GCC에 의해서 선언되는 매크로다.

ARM 버전 2, 3의 코어에 대해서는 코드 섹션 ❸과 같이 인터럽트 비활성화를 한다.

코드 섹션 ❹는 텍스트 섹션의 시작이며, 레지스터 r1～r6, ip, sp를 아래와 같이 설정해준다.

```
LC0                 ->      r1
__bss_start         ->      r2
_end                ->      r3
zreladdr            ->      r4
_start              ->      r5
_got_start          ->      r6
_got_end            ->      ip
user_stack+4096     ->      sp
```

표 5-1에 있는 ip, sp는 GCC에서 사용되는 ARM 레지스터들의 명명법(mnemonic)이다. GCC에 사용되는 ARM 레지스터 이름과 쓰임은 표 5-1을 참조하기 바란다. 그 다음에 not_relocated 레이블로 점프한다.

:: 표 5-1 GCC에 의해 사용되는 ARM 레지스터 이름과 쓰임

레지스터 이름	레지스터 명명법	쓰임
r0	a1	• 함수에 전달되는 첫 인자 • 결과 값 레지스터 • 스크래치 레지스터
r1	a2	• 함수에 전달되는 두 번째 인자 • 스크래치 레지스터
r2	a3	• 함수에 전달되는 세 번째 인자 • 스크래치 레지스터
r3	a4	• 함수에 전달되는 네 번째 인자 • 스크래치 레지스터
r4	v1	• 레지스터 변수
r5	v2	• 레지스터 변수
r6	v3	• 레지스터 변수
r7	v4	• 레지스터 변수
r8	v5	• 레지스터 변수
r9	v6 rfp	• 레지스터 변수 – 실제 프레임 포인터
r10	sl	• 스택 리미트(stack limit)

:: 표 5-1 (계속)

레지스터 이름	레지스터 명명법	쓰임
r11	fp	• 프레임 포인터(frame pointer)
r12	ip	• 스크래치 레지스터(scratch register)
r13	sp	• 스택 포인터(stack pointer)
r14	lr	• 링크 레지스터(link register)
r15	pc	• 프로그램 카운터(program counter)

5.2 BSS 영역 초기화하기 – not_relocated 레이블

압축된 커널인 zImage의 압축을 해제하기 위한 준비 단계로 BSS 영역을 초기화해주고 캐시 활성화 및 동적 메모리 영역을 설정한다. 이러한 설정은 커널 압축 해제 시 꼭 필요한 요구사항이다. 코드 5-2로 원리를 살펴보자.

:: 코드 5-2 arch/arm/boot/compressed/head.S의 not_relocated 레이블

```
not_relocated:
    mov r0, #0
1:                                                          ❶
    str r0, [r2], #4                @ bss 영역 초기화
    str r0, [r2], #4
    str r0, [r2], #4
    str r0, [r2], #4
    cmp r2, r3                      @ r2, r3를 비교
    blo 1b                          @ r2가 r3보다 작다면 뒤쪽 레이블 1로 점프

    bl  cache_on                                            ❷
    mov r1, sp                      @ malloc 영역은 스택 바로 위에 위치
    add r2, sp, #0x10000            @ 64k max, r2 = sp+ 0x10000
    ...
    bhs wont_overwrite              @ r4가 higher or same이라면 점프        ❸
```

코드 섹션 ❶은 BSS 영역을 0으로 초기화하며, BSS 영역은 초기화되지 않은 전역 데이터를 위한 공간이다. BSS 영역은 __bss_start와 _end 사이의 영역이며, 링커 스크립트 파일에 의해서 그 값이 결정된다.

코드 섹션 ❷를 통해서 cache_on을 호출하여 캐시를 활성화하고, 압축된 커널을 풀기 위한 동적 메모리 공간의 시작 주소(user_stack + 4KB)와 끝 주소(user_stack + 4KB + 64KB)

를 레지스터 r1, r2에 구한다. 즉, 동적 메모리 공간은 64KB다. 이것으로 커널 압축 해제를 위한 모든 준비가 마무리되었다. cache_on 레이블에 해당하는 코드 분석은 다음 절(5.3절)에서 자세히 알아볼 것이다.

코드 섹션 ❸의 wont_overwrite 레이블로 점프하여 본격적인 커널 압축 해제를 실행하게 된다. wont_overwrite에 대해서는 6.1절에서 알아볼 예정이다.

알아봅시다! vmlinux.lds 링커 스크립트 파일

링커 스크립트 파일은 링커에 의해 참조되며, Makefile에 의해 생성된 오브젝트 파일들을 어떤 식으로 링크하여 커널 이미지를 생성할지에 대한 정보를 가지고 있다. 링커는 항상 링커 스크립트를 사용한다. 만약에 링커 스크립트를 직접 제공하지 않으면 기본 스크립트를 이용한다. 기본 스크립트는 ld –verbose를 통해 확인할 수 있다.

```
code@kernel:~$ ld -verbose
GNU ld (GNU Binutils for Debian) 2.20.1.20100303
  Supported emulations:
   armelf_linux_eabi
   armelfb_linux_eabi
   elf_x86_64
   elf_i386
   i386linux
   elf_l1om
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
              "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SEARCH_DIR("/usr/arm-linux-gnueabi/lib");
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x00008000)); . =
SEGMENT_START("text-segment", 0x00008000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }

  ...

  .text           :
  {
    *(.text.unlikely .text.*_unlikely)
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
```

```
    *(.gnu.warning)
    *(.glue_7t) *(.glue_7) *(.vfp11_veneer) *(.v4_bx)
  } =0

  ...

  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.
      FIXME: Why do we need it- When there is no .bss section, we don't
      pad the .data section.  */
   . = ALIGN(. != 0 - 32 / 8 : 1);
  }
  _bss_end__ = . ; __bss_end__ = . ;
  . = ALIGN(32 / 8);
  . = ALIGN(32 / 8);
  __end__ = . ;
  _end = .; PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);

  ...

}

==================================================
code@kernel:~$
```

–T 옵션을 통해서 자신만의 링커 스크립트를 링커에게 제공할 수 있다. 아래 코드는 간단한 링커 스크립트 파일이다.

```
OUTPUT_FORMAT(  "elf32-littlearm",          /* 실행할 바이너리를 elf */
                "elf32-littlearm",          /* 포멧으로 생성한다. */
                "elf32-littlearm")
OUTPUT_ARCH(arm)                            /* 아키텍처를 ARM으로 명시한다. */
ENTRY(_start)                               /* 시작 함수를 _start로 설정한다. */
SECTIONS
{
    . = 0x00000000;                         /* 이미지 시작 주소 - mmu 사용을 하지
                                            않을 경우 0xA000000과같이 물리 메모리의
                                            주소가 들어갈 수도 있다.*/

    . = ALIGN(4);
```

```
    .text : { *(.text) }                    /* 각 오브젝트(.o) 파일의 text 섹션
                                               (코드영역)들이 링커에 의해서 하나의 text
                                                섹션으로 재배치된다. */
    . = ALIGN(4);
    .rodata : { *(.rodata) }                /* 각 오브젝트(.o) 파일의 rodata 섹션
                                               (읽기전용 데이터 영역)들이 링커에 의해서
                                               하나의 rodata 섹션으로 재배치된다. */
    . = ALIGN(4);                           /* 각 오브젝트(.o) 파일의 data 섹션(데이터
    .data : { *(.data) }                       영역)들이 링커에 의해서 하나의 data
                                                섹션으로 재배치된다. */
    . = ALIGN(4);                           /* 각 오브젝트(.o) 파일의 got 섹션(global
    got : { *(.got) }                          offset table)들이 링커에 의해서 하나의 got
                                               섹션으로 재배치된다.*/
    . = ALIGN(4);                           /* 각 오브젝트(.o) 파일의 bss 섹션(초기화
    .bss : { *(.bss) }                         되지 않은 데이터 영역)들이 링커에 의해서
                                               하나의 bss 섹션으로 재배치된다. */
}
```

5.3 캐시 활성화하기 – cache_on 레이블

BSS 영역 초기화와 동적 메모리 영역 설정 등을 해주었다면 커널 압축 해제를 위한 마지막 준비 단계로서 캐시 켜기(cache on)를 수행한다. 리눅스에서 캐시 켜기는 ARM 아키텍처 버전마다 서로 다르게 구현된다. cache_on 레이블에서는 현재 시스템의 ARM 아키텍처 버전에 맞는 캐시 켜기 서브루틴을 찾아 호출한다. 또한 __setup_mmu라는 서브루틴을 호출하여 페이지 디렉터리를 초기화해준다. 코드 5-3을 통해 자세히 살펴보자.

:: 코드 5-3 arch/arm/boot/compressed/head.S의 cache_on 레이블

```
cache_on:   mov r3, #8                  @ cache_on 함수                          ❶
    b   call_cache_fn

    ...

call_cache_fn:
    adr   r12, proc_types               @ r12 = proc_types에 대한
                                        @ 테이블의 시작 주소                      ❷

#ifdef CONFIG_CPU_CP15
    mrc p15, 0, r6, c0, c0              @ get processor ID
#else
    ldr r6, =CONFIG_PROCESSOR_ID        @ r6에 프로세서 아이디를 얻어옴
#endif
1:  ldr r1, [r12, #0]                   @ core ID                                ❸
    ldr r2, [r12, #4]                   @ Mask
```

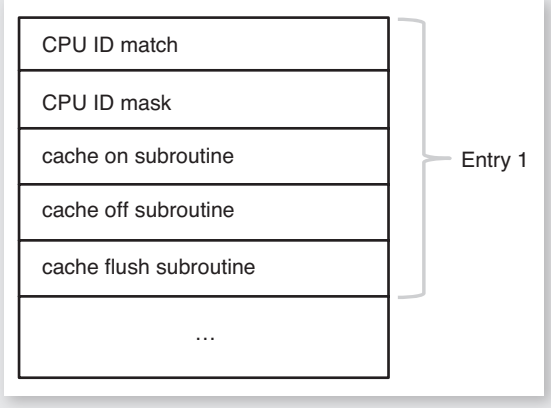
```
eor r1, r1, r6                  @ (real ^ match)-> exclusive or 연산함
                                @ 두 비트가 같으면 0, 두 비트가 다르면 1
tst r1, r2                      @ & mask    -> 마스크를 적용
                                @ tst는 선택된 bit들이 all zero인지 검사할 때 사용

addeq   pc, r12, r3             @ cache 함수 호출 -> 결과 값이 0이면
                                @ r12+r3 값을 pc에 넣어 점프.
add r12, r12, #4*5              @ 일치하지 않으면 r12에 20을 더함
                                @ (다음 엔트리를 가리키게 한다.)
b   1b                          @ 뒤에 있는 레이블 1로 점프
```

코드 섹션 ❶에서는 레지스터 r3에 상수 8을 저장하고 call_cache_fn 레이블로 점프한다. ARM 프로세서 버전마다 다른 캐시 켜기(cache on), 캐시 플러시(cache flush), 캐시 끄기(cache off) 기능을 proc_types 테이블로 관리한다. 즉, proc_types 테이블의 각 엔트리는 특정 ARM 프로세서 버전에 알맞은 캐시 켜기, 캐시 플러시, 캐시 끄기 서브루틴에 대한 정보를 갖고 있다. 상수 8은 proc_types 테이블의 각 엔트리에 존재하는 캐시 켜기 서브루틴의 시작 주소를 가리키는 오프셋으로 사용된다. proc_types 테이블의 구성은 "알아봅시다! proc_ types 테이블"을 참조하길 바란다.

알아봅시다! proc_types 테이블

proc_types 테이블은 ARM 프로세서 버전마다 하나의 엔트리로 표현되며, 엔트리는 아래 그림 5–2와 같이 구성된다.



:: 그림 5–2 **proc_types 테이블**

코드 5–3의 코드 섹션 ❸에서 ((real_id ^ match) & mask) == 0일 경우 엔트리가 일치한다. 그리고 라이트쓰루(writethrough, 바로쓰기) 정책을 사용하는 캐시는 캐시 켜기/끄기(cache on/off)에 대한 함수가 존재하지만, 라이트백(writeback, 나중쓰기) 정책을 사용하는 캐시의 경우는 메모리와의 동기화를 위해서 캐시 플러시(cache flush) 함수가 추가적으로 필요하다.

코드 섹션 ❷에서는 앞서 설명한 proc_types 테이블의 시작 주소를 레지스터 r12에 저장한다. CONFIG_CPU_CP15는 코프로세서(coprocessor1 CP15)[13]의 존재 여부에 대한 커널 설정 매크로다. 만약 CP15가 존재하는 ARM 아키텍처의 경우는 CPU ID를 CP15를 통해서 얻어온다.

코드 섹션 ❸에서는 현재 ARM 아키텍처에 알맞은 cache_on 서브루틴을 proc_types 테이블에서 찾은 후 점프한다.

앞서 proc_types 테이블을 통해서 ARM 프로세서 버전에 맞는 캐시 켜기 구현 서브루틴으로 이동해 왔다. ARM 버전별로 캐시 켜기 구현 방법은 서로 다르지만, 결국 하는 일은 모두 같다. 코드 5-4는 ARM v6의 예제다. ARM v6은 ARM v4와 같은 방법으로 캐시 켜기를 수행하므로 __armv4_mmu_cache_on 서브루틴을 수행한다.

:: 코드 5-4 arch/arm/boot/compressed/head.S의 __amrv4_mmu_cache_on 레이블

```
__armv4_mmu_cache_on:
    mov r12, lr                                          ❶

    bl  __setup_mmu
    mov r0, #0

    mcr p15, 0, r0, c7, c10, 4                           ❷
    mcr p15, 0, r0, c8, c7, 0
    mrc p15, 0, r0, c1, c0, 0

    orr r0, r0, #0x5000                                  ❸
    rr r0, r0, #0x0030

    bl  __common_mmu_cache_on                            ❹

    mov r0, #0                                           ❺
    mcr p15, 0, r0, c8, c7, 0

    mov pc, r12
```

먼저 코드 섹션 ❶에서 레지스터 lr에 담긴 복귀 주소를 r12에 저장한 후에 __setup_mmu 서브루틴을 호출한다. 이는 __setup_mmu 서브루틴 수행을 마치고 반환된 후 나머지 코드를 마저 수행하기 위해서다.

코드 섹션 ❷에서는 CP15를 이용해서 쓰기 버퍼의 내용을 모두 메모리에 업데이트하고,

13) 코프로세서는 추가된 명령어 집합(instruction set)과 설정 레지스터(configuration register)를 통해서 ARM 코어의 추가 기능을 지원한다. 특히 CP15는 ARM 프로세서가 Cache, TCM, Identification Register, Write Buffer, Memory Management를 제어할 수 있게 해준다.

I-Cache(명령어 캐시), D-Cache(데이터 캐시), TLB에 대해서도 플러시해준다.

코드 섹션 ❸에서는 CP15의 제어 레지스터(control register)의 내용을 읽어와서 I-Cache 활성화 비트, 라운드로빈 캐시 교체 정책 활성화 비트를 설정한다. 보다 자세한 내용은 "알아봅시다! CP15 제어 레지스터"를 참조하길 바란다. 변경된 설정에 대한 적용은 __common_mmu_cache_on 서브루틴에서 수행된다.

알아봅시다! CP15 제어 레지스터

ARM은 CP15의 제어 레지스터를 통해서 MMU와 캐시의 활성화, 캐시 교체 정책 설정 등을 수행한다. CP15의 제어 레지스터는 그림 5-3과 같다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SBZ		FA	TR	SBZ		EE	VE	XP	U	FI	SBZ		IT	SBZ	DT	L4	RR	V	I	Z	F	R	S	B	SBO			W	C	A	M

:: 그림 5-3 CP15 제어 레지스터

0x5000은 12번째 비트(I bit)와 14번째 비트(RR bit)를 설정함으로써 I-Cache 활성화와 라운드로빈 캐시 교체 정책을 설정한다.

코드 섹션 ❹의 __common_mmu_cache_on 서브 루틴에서는 앞서 코드 섹션 ❸에서 변경된 도메인(domain) 설정과 I-Cache 활성화 및 라운드로빈 캐시 교체 정책을 적용한다. 또한 페이지 디렉터리(page directory)의 시작 주소 값을 CP15의 전용 레지스터(Translation Table Base Register)[14]에 저장한다

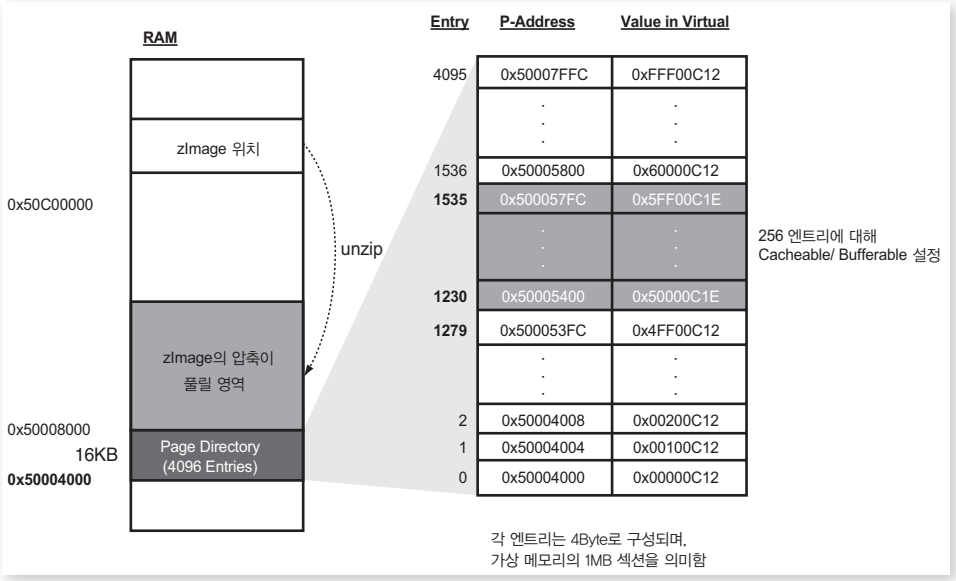
마지막으로 코드 섹션 ❺를 통해서 I-Cache, D-Cache, TLB를 플러시하고 코드 5-2의 bl cache_on 다음 명령어로 반환한다.

5.4 페이지 디렉터리 엔트리 초기화하기 – __setup_mmu 레이블

__setup_mmu 레이블은 cache_on 레이블 내에서 호출되며, 커널 압축 해제에 필요한 페이지 디렉터리 엔트리(page directory entry)를 초기화해준다. 특히 메모리의 256MB 영역에 대해서는 cacheable, bufferable로 설정을 해준다. 이것은 커널의 압축 해제 시 캐시와 쓰

14) TTBR는 L1(레벨 1) 테이블의 물리 주소를 저장하는 CP15 내의 특수 레지스터다. TTBR은 OS를 통해 사용되며, L1 테이블에 빠른 접근을 하기 위한 목적을 갖는다.

기 버퍼 사용을 통해서 압축 해제의 성능을 높이기 위함이다. __setup_mmu 수행 이후 페이지 디렉터리의 모습은 그림 5-4와 같다.



:: 그림 5-4 __setup_mmu 수행 후 페이지 테이블

코드 5-5를 통해서 커널이 어떻게 페이지 디렉터리를 초기화하는지 알아보도록 하자.

:: 코드 5-5 arch/arm/boot/compressed/head.S의 __setup_mmu 레이블

```
__setup_mmu:
    sub r3, r4, #16384                                          ❶
    bic r3, r3, #0xff              @ 포인터 정렬
    bic r3, r3, #0x3f00
    mov r0, r3                     @ r3 = 0x50004000 -> page directory 시작 주소

    mov r9, r0, lsr #18            @ 18비트만큼 right shift,                ❷
    mov r9, r9, lsl #18            @ 다시 18비트만큼 left shift ->r9 = start of RAM
    add r10, r9, #0x10000000       @ r10 = end of RAM

    mov r1, #0x12                  @ r1과 0xC00을 or 연산 -> r1 = 0xC12
    orr r1, r1, #3 << 10           @ 3 * 2^10 -> 3k -> 0xC00

    /* r2와 r4는 같음(zreladdr) */
    add r2, r3, #16384
1:  cmp r1, r9
```

```
    orrhs   r1, r1, #0x0c          @ r1의 값이 램 영역일 경우 set cacheable, bufferable,
    cmp r1, r10
    bichs   r1, r1, #0x0c          @ r1의 값이 램 영역이 아닐 경우 clear cacheable, bufferable

    str r1, [r0], #4               @ 1:1 mapping

    add r1, r1, #1048576

    teq r0, r2                     @ 두 값이 같은지 비교
    bne 1b                         @ 같지 않다면 뒤쪽에 있는 레이블 1로 점프

    ...

    mov pc, lr                     @ lr은 __arm4_mmu_cache_on 레이블에서
                                   @ bl __setup_mmu 다음 주소
ENDPROC(__setup_mmu)
```

코드 섹션 ❶에서는 페이지 디렉터리의 시작 주소를 레지스터 r0에 저장한다. 이 시작 주소는 ZRELADDR(zImage의 물리 주소) – 16KB를 통해서 구해지며, 커널 이미지 바로 앞에 위치하게 된다. 또한 16KB의 크기를 갖는다. ZRELADDR의 값은 각 아키텍처마다 달라지며, arch/arm/$(MACH)/Makefile.boot에서 찾아볼 수 있다. 페이지 디렉터리에 대해서는 "알아봅시다! MMU 페이지 테이블"을 참조하길 바란다.

코드 섹션 ❷를 통해서 페이지 디렉터리는 메모리를 1MB 단위의 섹션(section)으로 관리한다. 페이지 디렉터리 내의 4,096개 엔트리에 대해서 접근 권한(access permission)을 읽기/쓰기 가능으로 해준다. 또한 4,096개의 엔트리 중 256개(256MB 영역)에 대해서는 cacheable, bufferable 설정을 해준다.

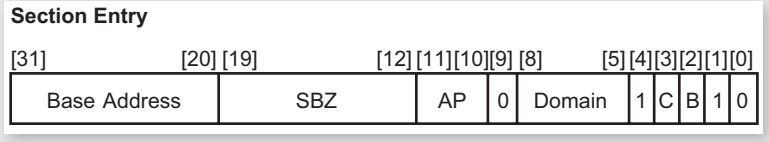
알아봅시다! MMU 페이지 테이블

MMU는 가상 주소를 물리 주소로 변환(virtual to physical translation), 메모리 접근 제어(memory access permission control), 메모리 내의 각 페이지에 대한 캐시 및 쓰기 버퍼 설정 등을 한다. 만약 MMU가 비활성화되어 있으면 가상 메모리와 물리 메모리는 1:1 매핑이 된다.

ARM MMU 하드웨어는 L1과 L2의 2단계 구조인 멀티레벨 페이지 테이블(multilevel page table) 구조를 갖는다. L1 페이지 테이블은 마스터 페이지 테이블(master page table)이라고 하고, 각 엔트리에는 L2 페이지 테이블의 시작 주소를 가지며, 1MB 크기의 페이지 변환을 한다.

이것 때문에 섹션 페이지 테이블(section page table)이라고 부르기도 한다. 마스터 L1 페이지 테이블은 4GB를 1MB의 섹션으로 나눔에 따라 4,096의 페이지 엔트리를 갖는다.

앞서 소개된 코드 5-5 상에서는 마스터 L1 페이지 테이블이 크기가 1MB인 가상 페이지의 시작 주소를 갖는 섹션 페이지 테이블의 역할을 한다. 리눅스에서 사용하는 섹션 테이블(section table)의 각 섹션 엔트리(section entry)는 그림 5-5와 같다.



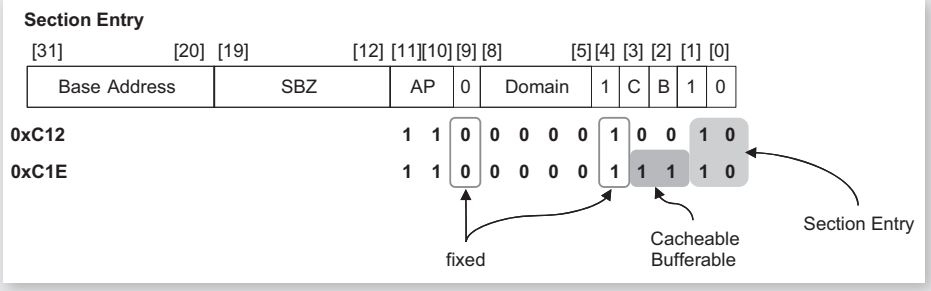
:: 그림 5-5 **MMU 섹션 테이블 엔트리**

알아봅시다! 캐시 및 쓰기 버퍼

프로세서에서 쓰기를 수행하는 속도와 메모리에서 쓰기를 처리하는 속도는 매우 큰 차이를 나타낸다. 프로세서에서의 쓰기 명령이 완료될 때까지 프로세서는 기다려야 한다. 메모리 쓰기는 프로세서 입장에서 매우 많은 클럭(clock)을 소모해야 하는 일이기 때문에 쓰기 버퍼를 통해서 이러한 지연이 발생하는 것을 어느 정도 대비한다. 쓰기 버퍼(write buffer)란 매우 작고 빠른 FIFO 메모리 버퍼이며, 메인 메모리에 쓰여져야 하는 데이터를 일시적으로 저장함으로써 프로세서가 메모리에 쓰기 동작 시 필요한 클럭 소모를 줄여준다.

마찬가지로 프로세서에서 읽기를 수행하는 속도와 메모리에서 읽기를 처리하는 속도는 매우 큰 차이를 나타낸다. 프로세서에서의 읽기 명령이 완료될 때까지 프로세서는 기다려야 한다. 캐시(cache)는 매우 빠른 속도의 저장장치이며, 임시적으로 프로세서가 메모리로부터 읽어온 코드 혹은 데이터를 저장한다. 이를 통해서 프로세서가 최근에 한 번 읽었던 데이터에 접근할 때 메모리 접근에 필요한 클록 소모를 줄여준다.

앞서 소개한 코드 5-5에서 256개의 엔트리에 대해서 cacheable, bufferable 설정을 하는 것을 보았다. 이를 통해서 256개의 섹션 내의 데이터들의 읽기/쓰기 수행 시 캐시와 쓰기 버퍼가 사용된다. 섹션 엔트리 내에 설정되는 값은 그림 5-6과 같다.



:: 그림 5-6 **MMU 섹션 엔트리에 설정되는 값들**

5.5 I-Cache 활성화 및 캐시 정책 적용하기 – __common_mmu_cache_on 레이블

__common_mmu_cache_on은 cache_on 레이블 내에서 호출되며, ARM 버전 7 이전의 프로세서에 대해서만 수행된다. I-Cache 활성화 및 캐시 교체 정책을 CP15에 적용하고, TTBR을 초기화한다. 코드 5-6을 통해 커널이 어떻게 I-Cache를 활성화하고 캐시 정책을 적용하는지 알아보도록 하자.

:: 코드 5-6 arch/arm/boot/compressed/head.S의 __common_mmu_cache_on 레이블

```
__common_mmu_cache_on:

    ...

    mcr p15, 0, r3, c2, c0, 0          @ 페이지 테이블 주소를 저장          ❶
    ...
1:  mcr p15, 0, r0, c1, c0, 0          @ r0을 컨트롤 레지스터에 넣기        ❷
    mrc p15, 0, r0, c1, c0, 0
    sub pc, lr, r0, lsr #32
```

우선 코드 섹션 ❶에서는 레지스터 r3에 저장되어 있는 페이지 디렉터리의 주소 값을 CP15에 있는 TTBR에 저장한다.

코드 섹션 ❷에서는 __armv4_mmu_cache_on 레이블에서 설정해준 라운드로빈 캐시 교체 정책, I-Cache 활성화 설정을 CP15의 제어 레지스터(control register)에 적용한다.