

Linux Kernel Analysis for ARM

2011년 5월 22일

Version 0.6

iamroot Kernel 6차 Members

Prologue

iamroot.org를 통해 2009년 5월 시작한 분석이 2011년 3월에 끝났다. 끝냈다고 보다는 한번 봤다는 표현이 더 정확할 것이다. 커널 분석을 시작할 때는 자신도 만만했는데...분석하면 할 수록 “제대로 아는 것이 하나도 없구나” 하는 확인 작업이 되었고, 급기야 우리를 한없이 작게 만들어 버렸다.

많이 부족하다! 그리고 채워야할 것이 너무나 많다! ... 하지만 분석하면서 고민한 내용을 여러분들과 공유하고자 한다.

우리가 공유 하는 것은 우리가 아는 것이 많아서가 아니다. 이해하고 있는 지식이 결코 옳다고 믿어서도 아니다. 단 하나, 우리는 리눅스를 공부했기 때문이다! 1991년 리누스가 수줍은 이메일로 자신의 커널을 세상에 공유했던 것 처럼...우리는 누군가에게 이문서가 도움이 되길 바랄 뿐이고, 또 다른 누군가는 이 문서를 완성해주길 바랄 뿐이다. 그래서 결국 우리가 여러분의 도움으로 커널을 더 잘 이해하고 싶은 마음 뿐이다.

iamroot Kernel 6차 멤버

노서영, 윤석훈, 이윤재, 송원준, 강진성, 안정모, 임윤재

문서의 배포

이 문서는 자유로운 복사와 배포가 가능합니다. 단, 상업적 목적의 복사와 배포는 금합니다. 문서의 일부를 사용할 때에는 출처를 밝혀주시기 바랍니다. 문서에 사용된 내용이 우리의 의도와는 다르게 저작권법에 위배가 된다면 삭제하겠습니다. 또한 문서에 대한 의견과 도움을 주시려면 아래 이메일로 보내 주시기 바랍니다. 다음 버전에 반영하여 공유하도록 하겠습니다.

노서영	seoyoungnoh@gmail.com
윤석훈	mindwave@nate.com
이운재	spanico@gmail.com
송원준	rootfriend@gmail.com
강진성	paranpi7@nate.com
안정모	jungmoan@gmail.com
임운재	launius@naver.com

커널버전

분석을 위해 사용한 커널 소스는 **2.6.30.4**이며 상위 버전의 커널소스와는 내용이 다를 수 있음.

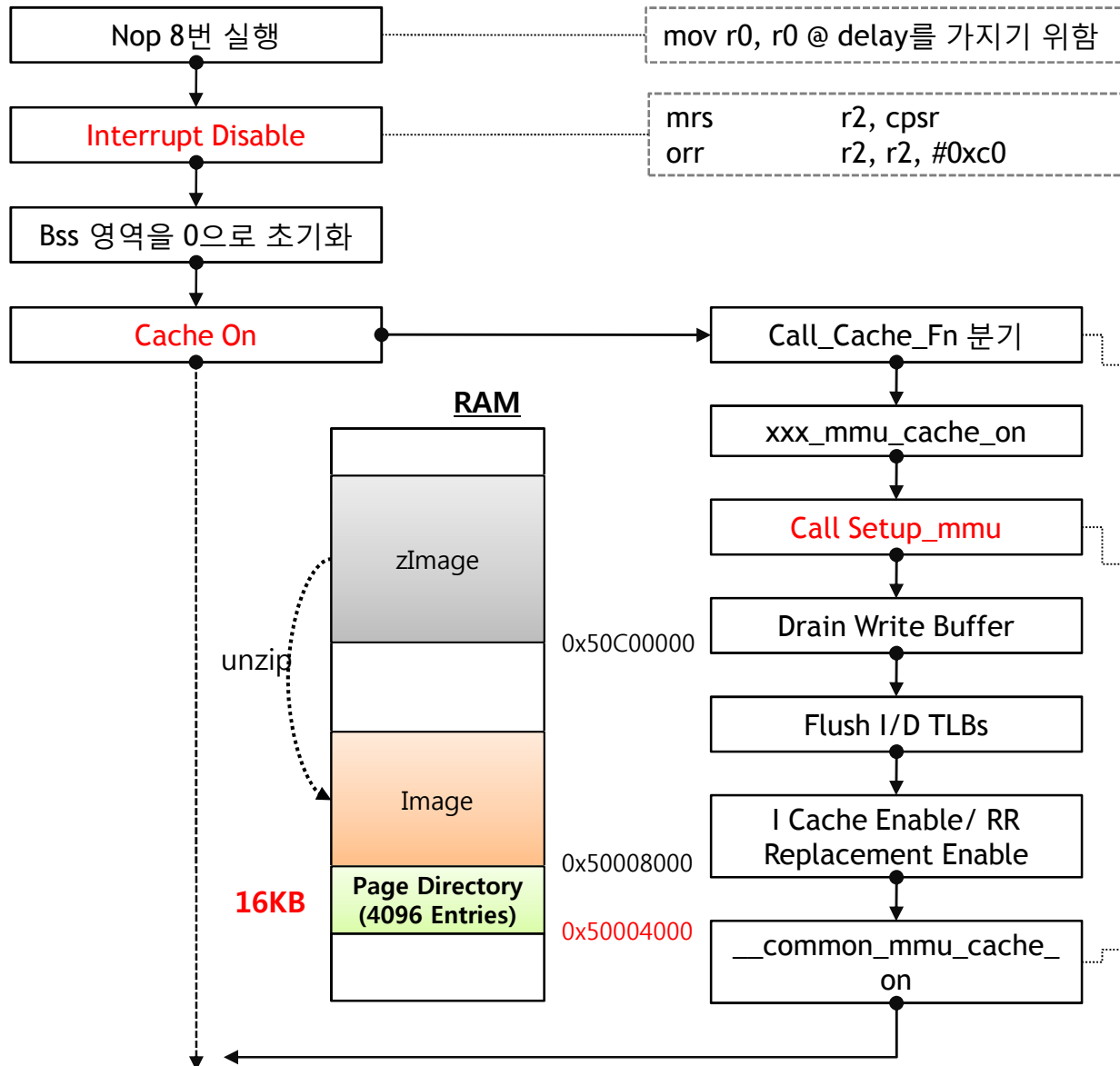
History

Date	Version	Description	Etc.
2009-12-17	0.1	Initially created	
2010-03-03	0.2	<ul style="list-style-type: none"> • setup_arch/bootmem_init/bootmem_free_node/free_area_init_node/free_area_init_core까지 정리 • _find_next_zero_bit_le 로직 정리 • Memory Bank, Zone, Node, NUMA, UMA • Memory Hot-Plug • ./Documentation/arm/Bootimg 정리 • ./Documentation/arm/Porting 정리 	
2010-11-13	0.3	<ul style="list-style-type: none"> • Thread Model 정리 • Ext2 File System 정리 • Preemption 정리 • Cache 	
2011-03-20	0.4	<ul style="list-style-type: none"> • Kernel Lock 정리 • Memory Management 보강 • Malloc 동작 방식 • Program Execution Step 	
2011-05-21	0.6	<ul style="list-style-type: none"> • 최종취합 및 Final Touch • First Release 	0.5 취합

compressed/head.S

U-Boot로부터 전달된 값

R0 ← 0/ R2 ← Architecture ID/ R3 ← Board Information (aTag)



Process Type Table

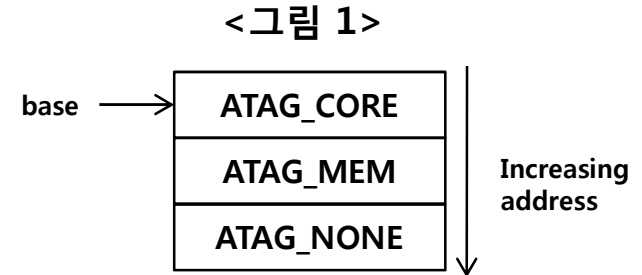
Offset

Architecture Number	0
Magic Number	4
xxx_mmu_cache_on	8
xxx_mmu_cache_off	12
xxx_mmu_cache_flush	16

Boot Loader

■ Boot Loader가 제공해야 하는 Functionalities

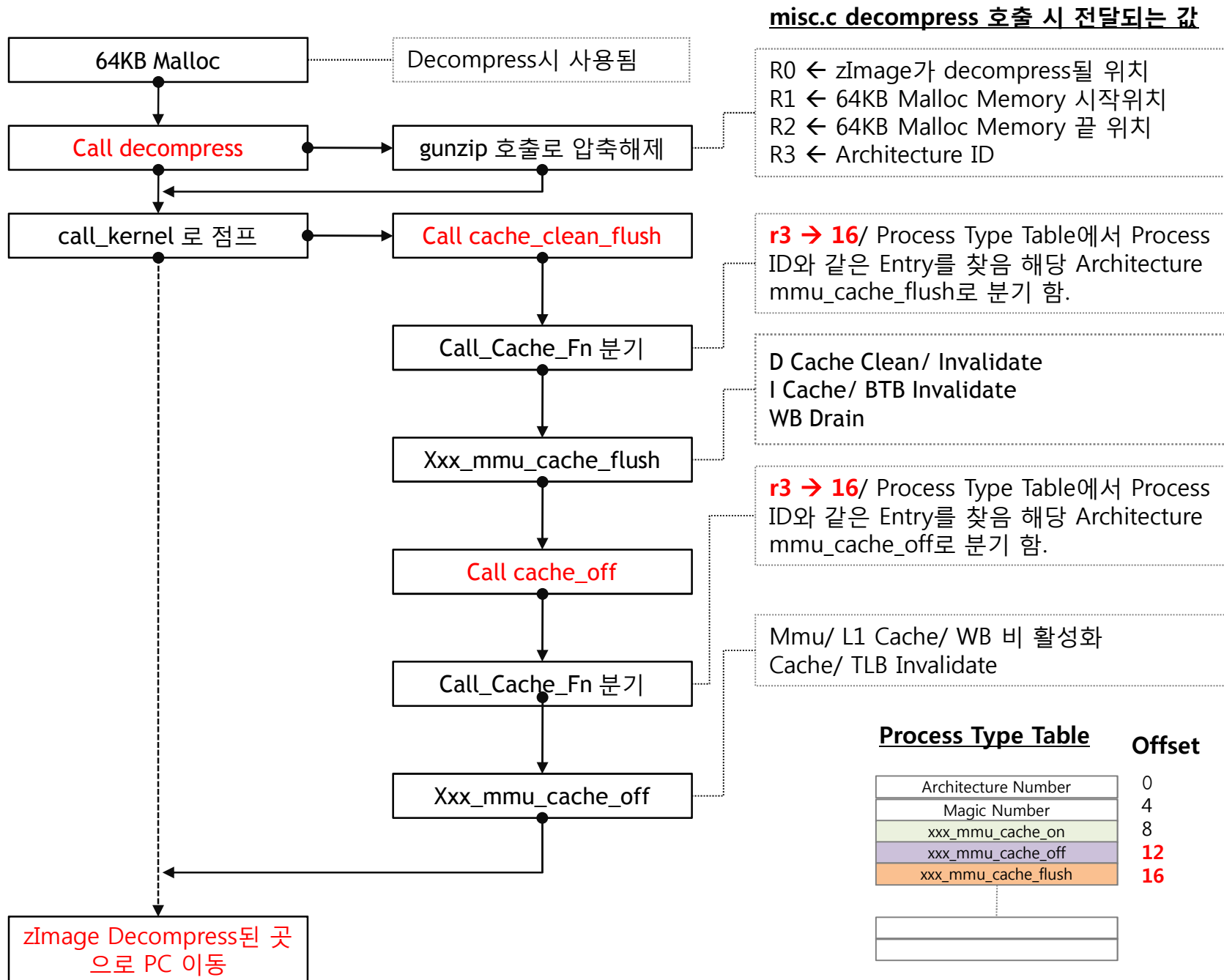
1. Setup and initialize the RAM.
2. Initialize one serial port
3. Detect the machine type.
- 4. Setup the kernel tagged list.**
5. Call the kernel image.



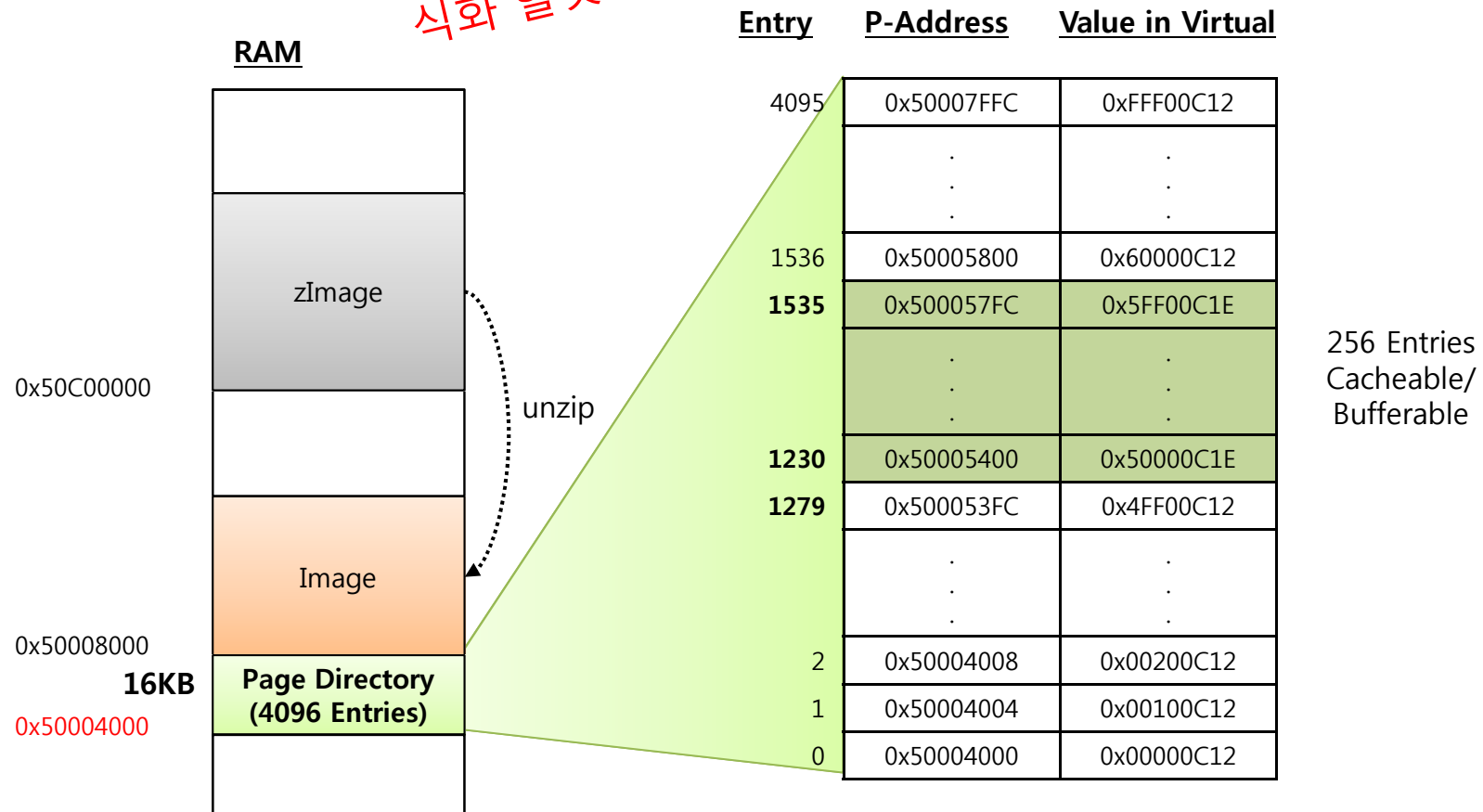
■ Setup the kernel tagged list

- 유효한 Tagged List는 **ATAG_CORE 로 시작해서 ATAG_NONE로 끝나야 함.**
- ATAG_CORE tag는 빈 공간일 수 있는데, 이 경우 ATAG_CORE tag의 size field는 '2' (0x00000002)로 셋팅 되어야 함.
- ATAG_NONE 의 size filed는 0으로 셋팅 되어야 함.
- Tagged List에 어떠한 수의 tag들이 올 수 있지만, 동일한 tag가 올 경우 append될 지 overwrite될지는 정의되지 않음.
- Boot Loader는 최소한 시스템 메모리의 위치와 크기, 루트파일 시스템 위치를 패스해야 하므로 최소한 Tagged List는 <그림 1>과 같음
- Tagged List는 메모리상에 위치해야 함.
- Tagged List는 Kernel의 Decompressor나 initrd 'bootp' 프로그램이 overwrite하지 않아야 함. **따라서 주로 RAM의 첫 16KB에 위치하게 됨.**

Reference: ./Documentation/arm/Booting



ToDo: Entry 4Byte의 구성을 도
식화 할것



각 Entry 4Byte로 구성되며 Virtual Memory의 1MB Section을 의미 함

Decompressor Symbols and Kernel Symbols

Reference: ./Documentation/arm/Porting

■ Decompressor Symbols

Symbols	Description
ZTEXTADDR	<ul style="list-style-type: none">Decompressor의 시작 주소.Decompressor code가 호출될 때에는 MMU가 off된 상태이기 때문에 virtual 및 physical address는 논의 대상이 아니다.Booting하기 위해서 ZTEXTADDR에 있는 Kernel을 호출하게 된다.
ZBSSADDR	<ul style="list-style-type: none">Decompressor가 사용할 0으로 초기화된 작업 영역의 시작 주소.Decompressor는 이 영역을 0으로 초기화 한다.
ZRELADDR	<ul style="list-style-type: none">압축이 해제된 Kernel이 쓰여질 주소<code>__virt_to_phys(TEXTADDR) == ZRELADDR</code>
INITRD_PHYS	<ul style="list-style-type: none">RAM disk가 위치할 물리주소이고 bootpImage를 이용할 경우만 관련됨.
INITRD_VIRT	<ul style="list-style-type: none">RAM disk의 가상주소<code>__virt_to_phys(INITRD_VIRT) == INITRD_PHYS</code>
PARAMS_PHYS	<ul style="list-style-type: none">para_struct 구조체나 tag_list의 물리주소로 커널에게 다양한 실행 환경에 대한 파라미터를 전달함.

Decompressor Symbols and Kernel Symbols

Reference: ./Documentation/arm/Porting

■ Kernel Symbols

Symbols	Description
PHYS_OFFSET	<ul style="list-style-type: none">RAM의 첫번째 뱅크의 시작 물리주소
PAGE_OFFSET	<ul style="list-style-type: none">RAM의 첫번째 뱅크의 가상주소로 커널 부팅단계에서는 가상주소 PAGE_OFFSET이 물리주소 PHYS_OFFSET으로 매핑 되고 TASK_SIZE와 같은 값을 가져야 함.
TASK_SIZE	<ul style="list-style-type: none">User Process의 최대 크기 (바이트). Max Addr = User Process가 Access + 1User Process의 스택은 Max Addr.에서 아래로 자람.TASK_SIZE아래의 모든 가상 주소는 User Process 영역이며, 커널이 Process 기준으로 동적으로 관리함. 이 영역을 User Segment라고 부름. TASK_SIZE 위의 모든 가상 주소는 모든 Process에 공통이고 이 영역을 Kernel Segment라고 부름.
TEXTADDR	<ul style="list-style-type: none">커널의 가상 시작주소로 일반적으로 PAGE_OFFSET + 0x8000가 됨.이 주소는 커널 Image가 끝나는 위치
DATAADDR	<ul style="list-style-type: none">커널 데이터 세그먼트를 위한 가상주소. Decompressor를 사용할 때는 정의되면 안됨.
VMALLOC_START/ VMALLOC_END	<ul style="list-style-type: none">vmalloc() 영역에 대한 가상주소 Boundary이며 vmalloc이 이 영역에 대해 overwrite를 하기 때문에 이 영역에는 어떠한 Static Mapping이 와서는 안됨이 가상주소들은 모두 Kernel Segment내에 존재하며 일반적으로 vmalloc() 영역은 변수 high_memory를 이용해서 찾게 되는 마지막 가상 RAM 주소보다 위쪽 바이트인 VMALLOC_OFFSET 바이트에서 시작함.
VMALLOC_OFFSET	<ul style="list-style-type: none">가상 RAM과 vmalloc 영역 사이의 공간 (hole)을 제공하기위해서 VMALLOC_START와 관계된 Offset로 이렇게 하는 이유는 Out of Bound 메모리 접근하는 것을 Catch하기 위함. 일반적으로 8MB로 세팅됨.

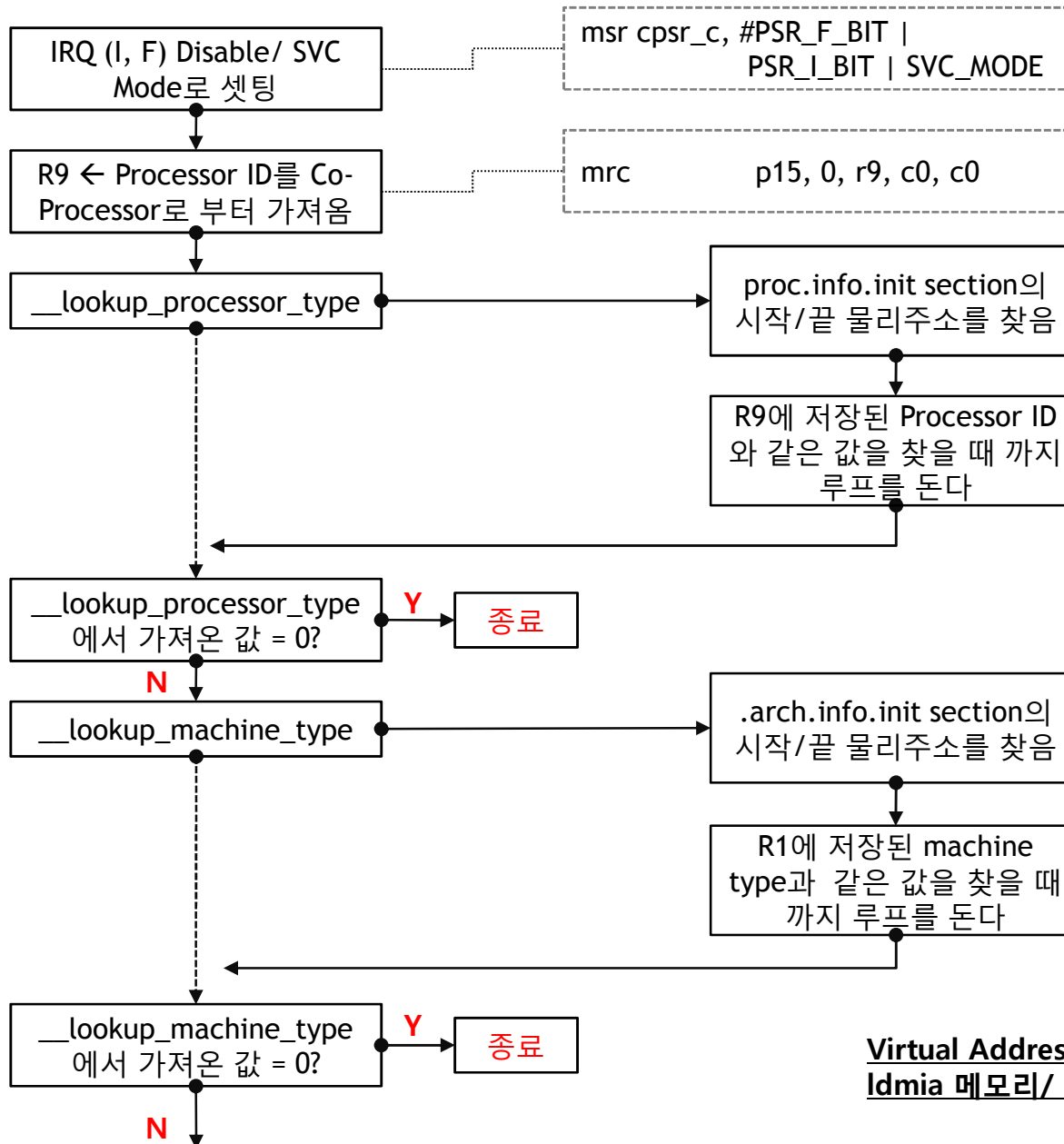
Decompressor Symbols and Kernel Symbols

Reference: [./Documentation/arm/Porting](#)

■ Architecture Specific Macros

Symbols	Description
BOOT_MEM(pram,pio,vio)	<ul style="list-style-type: none">• Debugging 영역이 Architecture 종속적인 코드에 의해 초기화됨.• 'pram'은 RAM의 시작 물리주소를 나타내고 반드시 Present해야 하고 PHYS_OFFSET과 같아야 함.• 'pio'는 IO를 포함하고 있는 8MB의 물리주소이고 arch/arm/kernel/debug-armv.S에 있는 debugging macro들과 함께 사용.• 'vio'는 8MB Debugging 영역의 가상주소.
BOOT_PARAMS	<ul style="list-style-type: none">• PARAMS_PHYS와 같음.
FIXUP(func)	<ul style="list-style-type: none">• Memory subsystem이 초기화전에 실행되는 Machine 종속적인 fixups
MAPIO(func)	<ul style="list-style-type: none">• Map IO 영역에 대한 Machine 종속적인 함수
INITIRQ(func)	<ul style="list-style-type: none">• Interrupt들을 초기화하기 위한 Machine 종속적인 함수

kernel/head.S,



msr cpsr_c, #PSR_F_BIT |
PSR_I_BIT | SVC_MODE

mrc p15, 0, r9, c0, c0

proc.info.init section의
시작/끝 물리주소를 찾음

R9에 저장된 Processor ID
와 같은 값을 찾을 때 까지
루프를 돈다

종료

.arch.info.init section의
시작/끝 물리주소를 찾음

R1에 저장된 machine
type과 같은 값을 찾을 때
까지 루프를 돈다

종료

head.S/ head-common.S에서 사용하는 값

R1 : machine type
R2 : aTag Pointer
R9: Processor ID

head-common.S

_proc_info_begin/
_proc_info_end 의 물리주소

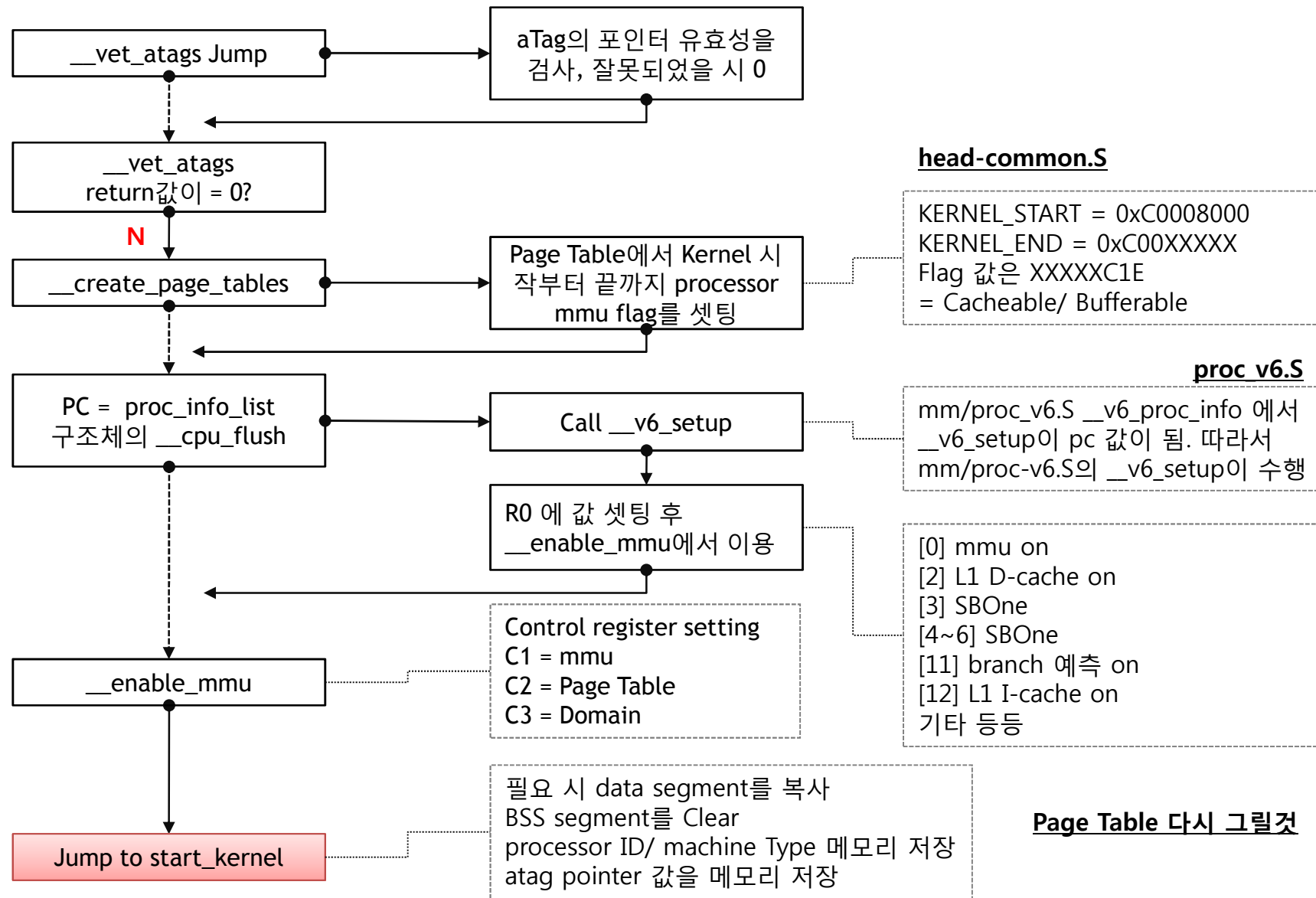
sizeof(proc_info_list) 단위
로 .proc.info.init section의 memory
를 access 함

_arch_info_begin /
_arch_info_end 의 물리주소

sizeof(machine_desc) 단위
로 .arch.info.init section의
memory를 access 함

Virtual Address -> Physical Address로 전환하는 방법
Idmia 메모리/ 레지스터지정 방법

kernel/head.S

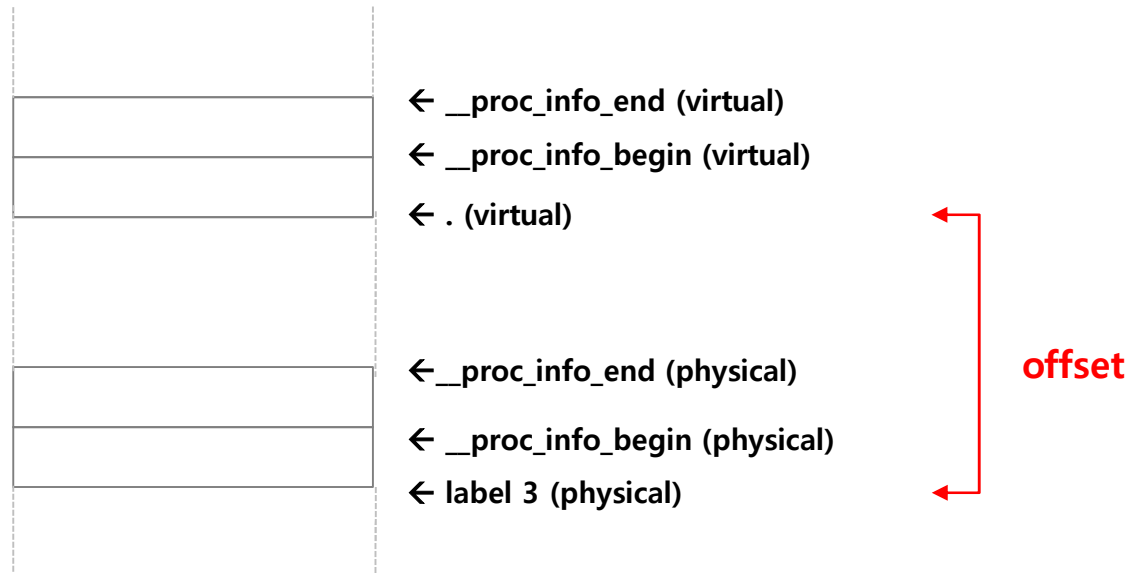


Virtual Address -> Physical Address로 전환: Idmia 메모리/ 레지스터지정 방법

ldmda, ldmia, ldmdb, ldmda 명령어의 경우 높은 번호의 Register가 항상 높은 메모리와 매핑 됨. 따라서, LDMDA R0, { R1-R4 } 는 $R4 \leftarrow R0$, $R3 \leftarrow R0 - 4$, 가 된다.

head-common.S 에서

프로세서 타입을 찾는 부분 `__lookup_processor_type` 및 머신타입을 찾는 부분 `__lookup_machine_type`이 아래와 같은 동일한 방법으로 가상주소를 물리주소로 전환하여 찾음

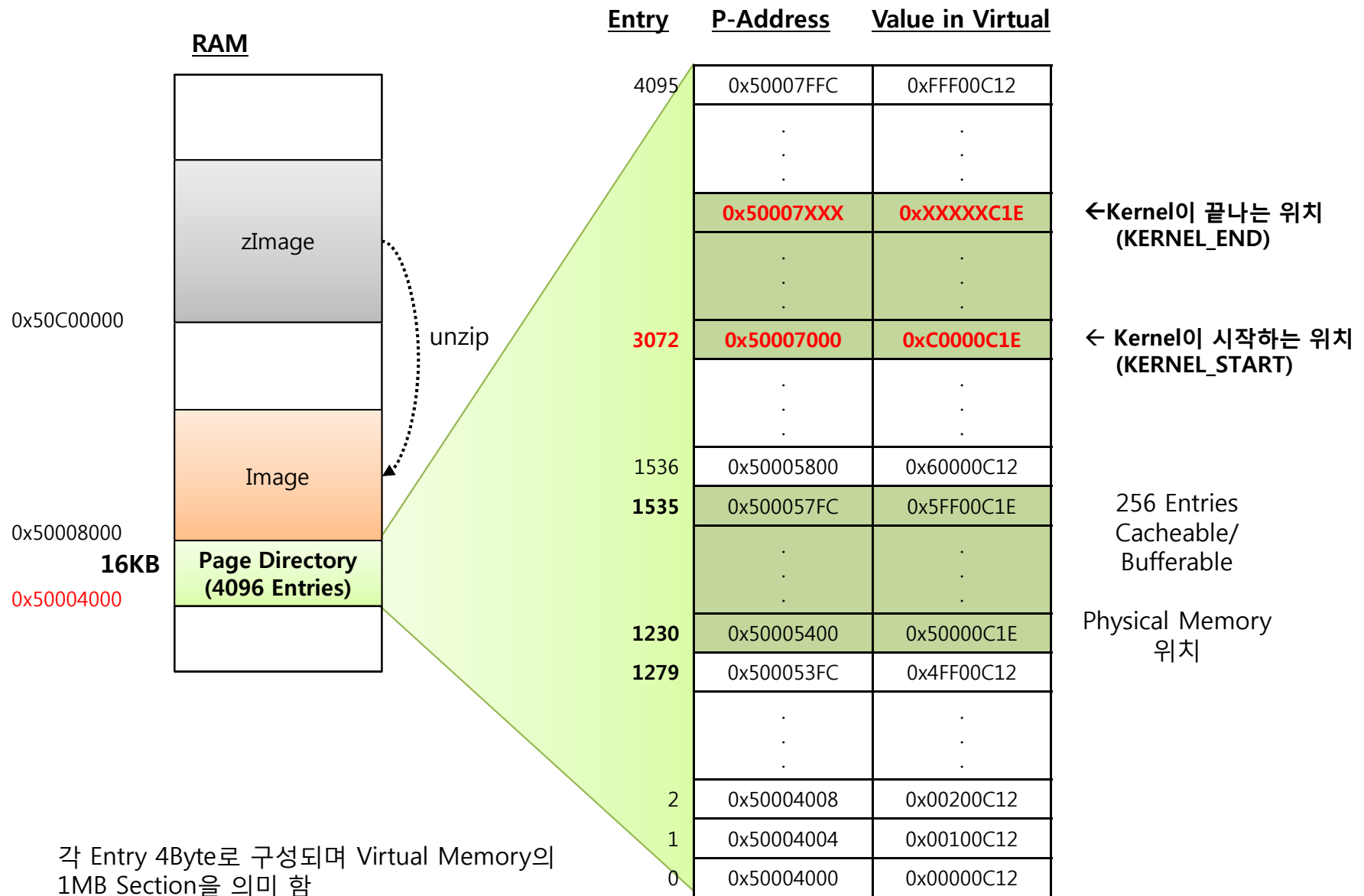


$$\text{label 3 (physical)} - \text{. (virtual)} = \text{offset}$$

$$\text{__proc_info_begin (virtual)} + \text{offset} = \text{__proc_info_begin (physical)}$$

Note: offset의 값은 음수가 될수 있음

kernel/head.S,의 create_page_tables 수행 후 Page Table 결과



init/main.c

asmlinkage를 통해 assem과 link될 수 있게 함 head-common.S에서 Call

start_kernel

Weak attribute를 사용하여 Arch가 sparc일 경우 수행 다른 arch일 경우 아무것도 하지 않음

CONFIG_DEBUG_OBJECTS가 정의된 경우 Kernel에서 관리하는 Object들에 대한 tracking이 가능하도록 하는 것 같음 (To Be Cleared)

control group은 resource tracking을 위한 것이며 cgroup/ subsystem/ hierarchy로 구성됨.
(./Documentation/cgroups/*.txt 참조)

CONFIG_TRACE_IRQ_FLAGS가 정의되어 있다면, early_boot_irqs_enabled = 0으로 만듦. Debugging시에 도움을 주기 위한 것으로 현재 early bootup code에 있고 invalid한 irq-on event에 대해 warning 메시지를 주기 위함.

smp_setup_processor_id()

lockdep_init

debug_objects_early_init

boot_init_stack_canary

cgroup_init_early

local_irq_disable

early_boot_irqs_off

early_init_irq_lock_class

CONFIG_LOCKDEP가 정의된 경우 Lock dependency tool이 사용하기 위한 자료 구조를 초기화 함.
Classhash_table ($2^{12} = 4096$ entry)
Chainhash_table ($2^{14} = 4 * 4096$ entry)

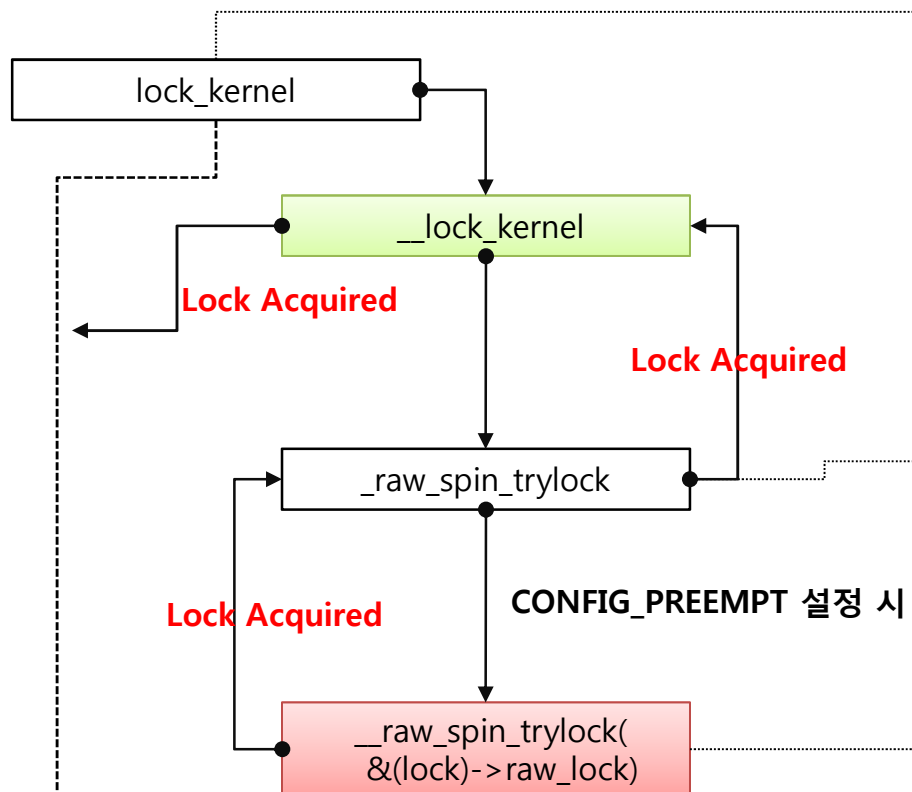
x86에서 사용됨. Stack에 특정 패턴을 넣어 함수가 return할 때 overwrite하는 것을 detection 함.
Canary 새가 광산에서 문제가 발생할 때 제일 먼저 detection한 것을 비유

cpsr register의 I bit를 mask 시킴:
cpsid I
current task의 hardirq를 disable 시킴

CONFIG_GENERIC_HARDIRQS가 정의된 경우 lockdep_init()에서 초기화 했던 classhash_table에 NR_IRQ만큼의 irq를, 즉 모든 irq_lock을 등록함

Kernel의 early stage로 주로 Debugging 관련된 셋팅을 함. 따라서 관련된 Define값들이 정의되지 않았을 경우, 대부분 아무 일도 수행하지 않음

Kernel Lock 획득하는 방법



```

#define __lockfunc __attribute__((section(".spinlock.text")))

void __lockfunc lock_kernel(void)
{
    int depth = current->lock_depth+1;
    if (likely(!depth))
        __lock_kernel();
    current->lock_depth = depth;
}

```

최초 `current->lock_depth`의 값은 -1이기 때문에 `!depth`가 true가 됨. True일 경우 `_lock_kernel` 호출

Lock을 획득하는데 성공했다면 즉 `_raw_spin_trylock`의 리턴값이 1이라면 리턴하고 그렇지 않다면, 스핀

arch/arm/include/asm/spinlock.h

```

static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    unsigned long tmp;

    __asm__ __volatile__(
        "1: ldrex    %0, [%1]Wn"
        "   teq     %0, #0Wn"
#ifdef CONFIG_CPU_32v6K
        "   wfeneWn"
#endif
        "   strexeq %0, %2, [%1]Wn"
        "   teqeq   %0, #0Wn"
        "   bne     1b"
        : "=&r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc");

    smp_mb();
}

```

Inline assembly code:

- lock->lock 값을 **배타적으로** tmp에 저장
- tmp = 0 인지 검사
 - True 라면, 배타적으로 lock->lock = 1 설정
 - tmp = 0인지 검사 (strexeq에 의해 결과값이 tmp에 저장됨)
- tmp != 0이라면
 - 1 번으로 분기 하여 루프를 돌

init/main.c

tick_init

struct notifier block

int (*notifier_call)
notifier_block *next
int priority

kernel/time/tick-common.c

```
static struct notifier_block tick_notifier =  
{ .notifier_call = tick_notify, }
```

tick_notify는 static 함수이며, clock 이벤트 디바이스들에 대한 Notification을 담당하며, 이벤트에는, CLOCK_EVT_NOTIFY_ADD, CLOCK_EVT_NOTIFY_BROADCAST_ON, CLOCK_EVT_NOTIFY_CPU_DYING, CLOCK_EVT_NOTIFY_CPU_DEAD, CLOCK_EVT_NOTIFY_SUSPEND 등이 있음

clockevents_register_notifier(
&tick_notifier)

spin_lock(
&clockevents_lock)

raw_notifier_chain_register(
&clockevents_chain,
tick_notifier)

spin_unlock
(&clockevents_lock)

clockevents_lock에 대한 spin lock

clockevents_chain에 tick_notifier를 등록함

삽입하고자 하는 tick_notifier의 priority와 clock_event_chain에 속한 notifier_block의 priority를 비교하여 clock_event_chain에 priority가 높은 순으로 삽입

clockevents_lock의 spin lock 해제

Notifier Chain의 4가지 Type

1. Atomic Notifier Chains

Atomic Notifier Chain Type은 Chain Callback 함수들이 Interrupt나 Atomic Context에서 동작하고 Callback 함수가 Block되는 것을 허용하지 않음.

2. Blocking Notifier Chains

Blocking Notifier Chain Type은 Callback 함수들이 프로세스 컨텍스트에서 수행되며, Callback 함수가 Block되는 것을 허용

3. Raw Notifier Chains

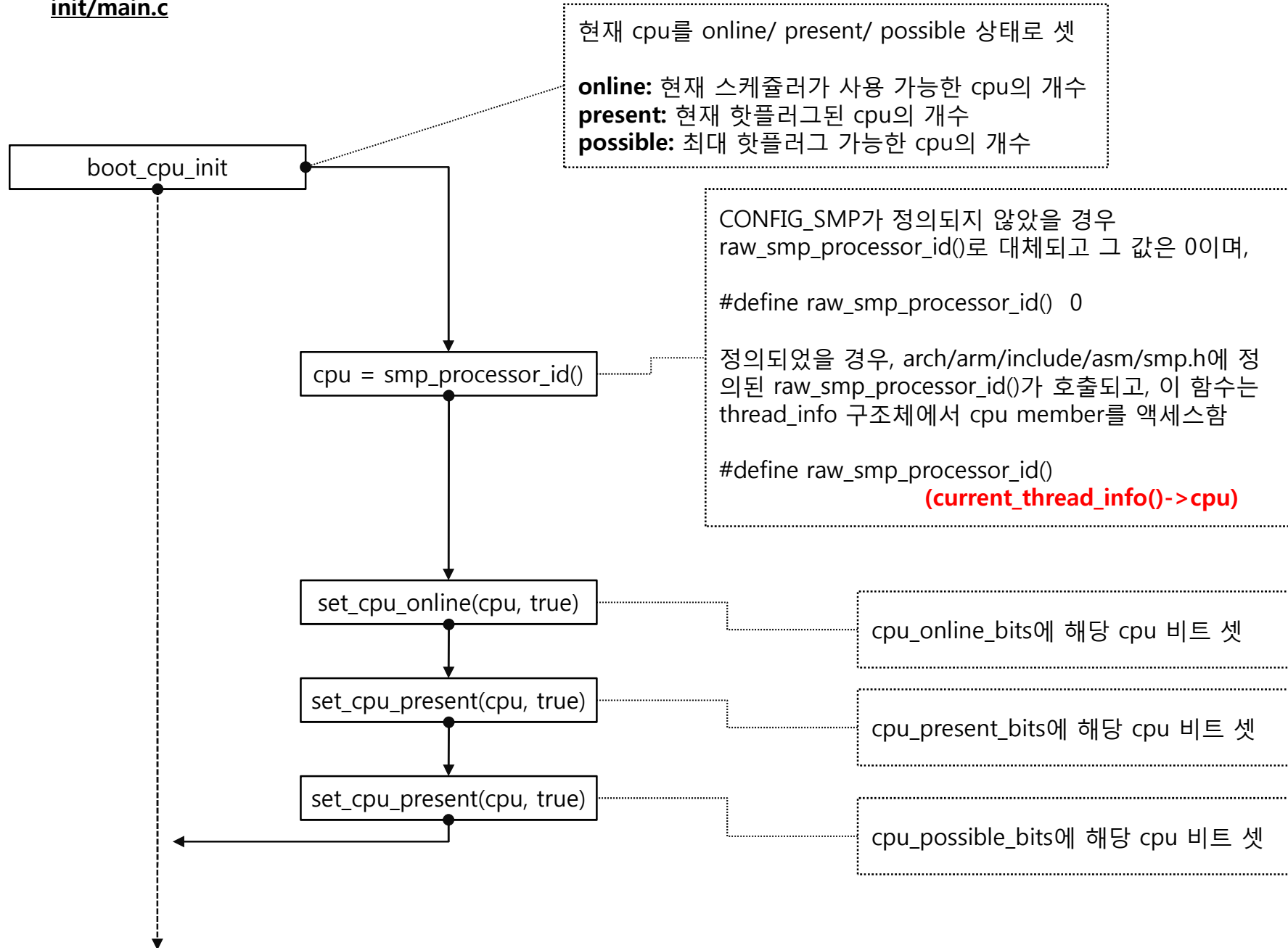
Raw Notifier Chain Type은 callback, registration, unregistration에 대한 제약이 전혀 없으며, 호출자 (Caller)에 의해서 모든 locking 및 protection이 제공되어야 함.

4. SRCU (Sleepable Read-Copy Update) Notifier Chains

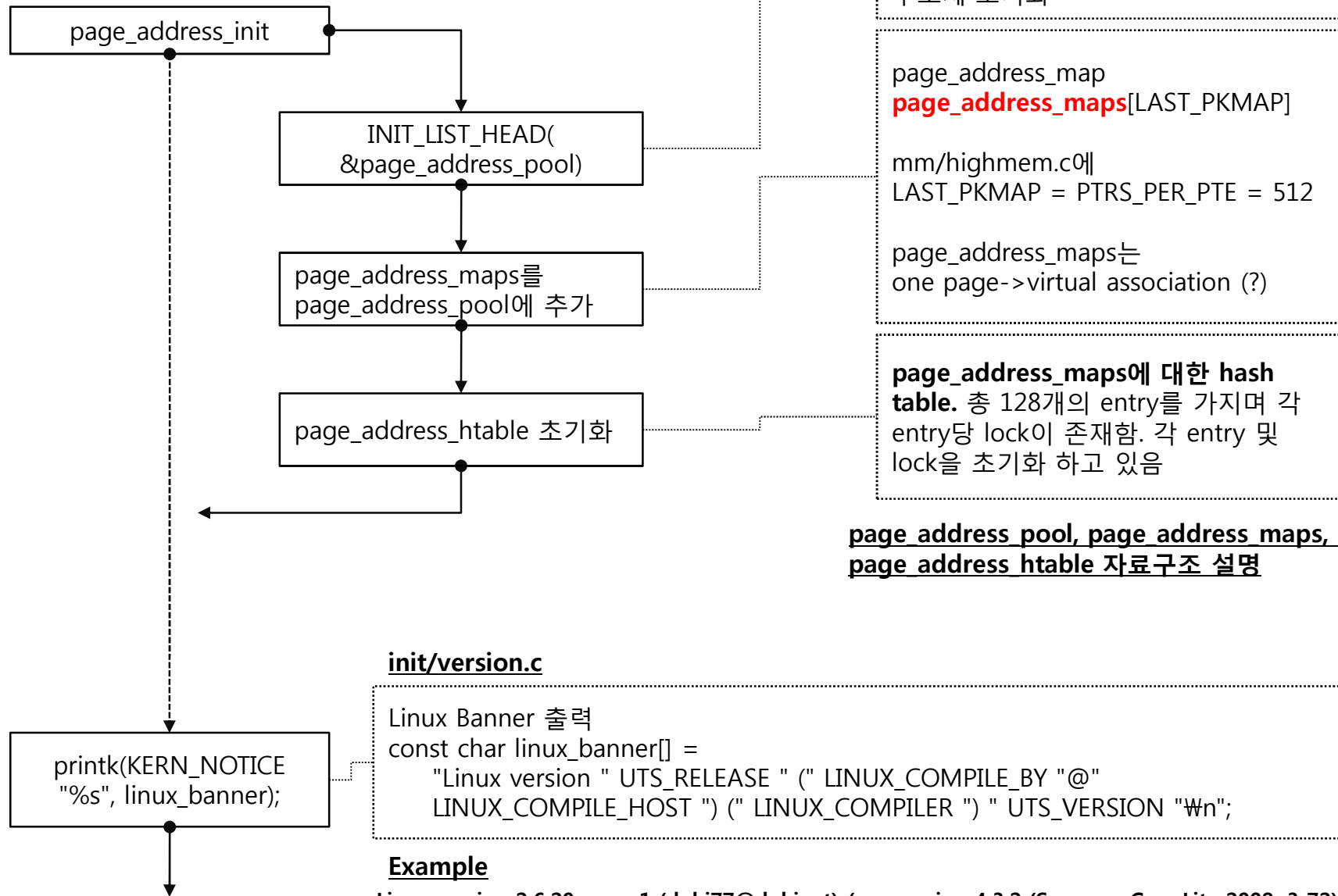
SRCU Notifier Chain Type은 Blocking Notifier Chain의 변종으로 동일한 제약을 가짐. srcu_notifier_call_chain() 함수는 SRCU (Sleepable Read-Copy Update)를 사용하며 SRCU는 rw-semaphore 보다 chain link들에 대한 보호에서 작은 오버헤드를 가짐 (no cache bounce & no memory barrier). 이에 대한 대가로 srcu_notifier_chain_unregister()는 조금의 오버헤드가 있음. SRCU Notifier Chain들은 Chain이 빈번하게 호출되지만 Notifier Block에서 거의 제거되지 않은 경우에 사용되어야 하며, 사용하기 위해서는 Runtime Initialization이 필요함.

chain을 등록 (register)할 때, atomic_notifier_chain_register(), blocking_notifier_chain_register(), srcu_notifier_chain_register()를 이용하며, atomic_notifier_chain_register()는 atomic 컨텍스트에서 호출되지만, 나머지 두개는 process 컨텍스트에서 호출되어야 함. chain을 해제 (unregister) 할 때는 call chain에서 unregister 함수가 호출되면 안됨

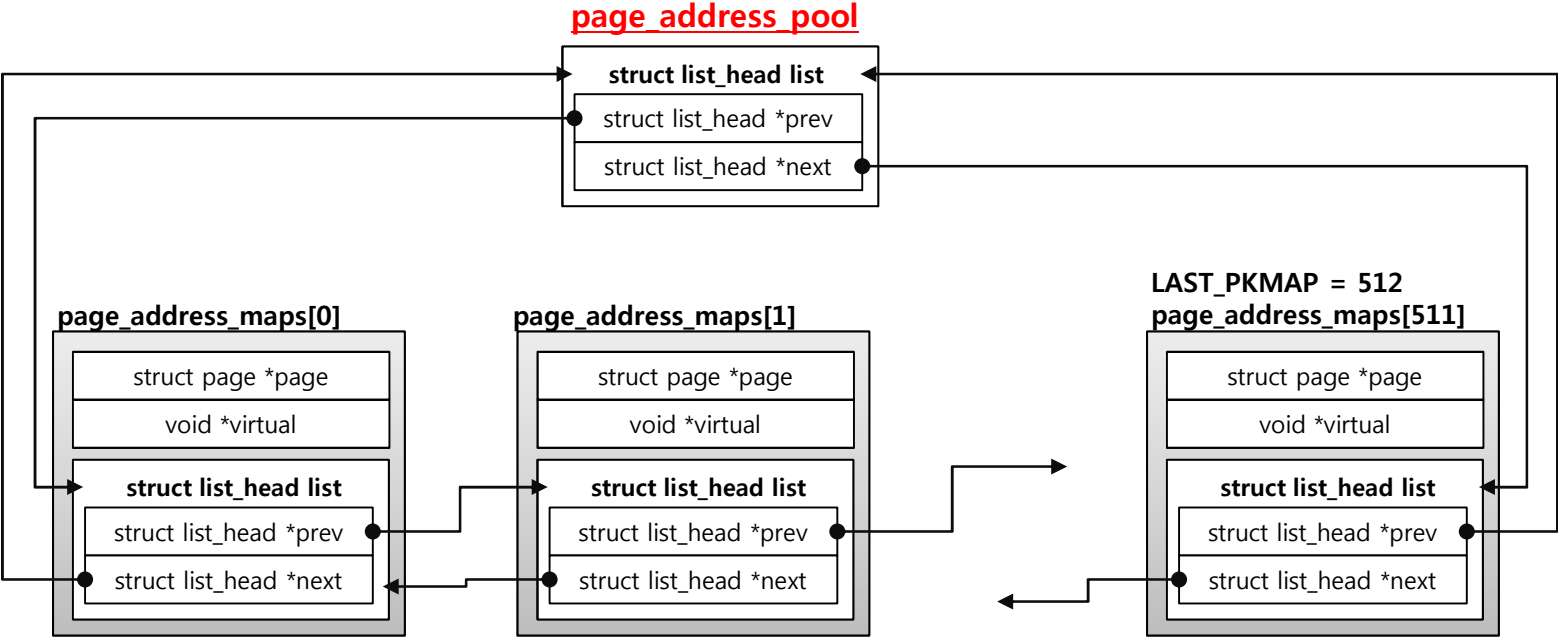
init/main.c



init/main.c

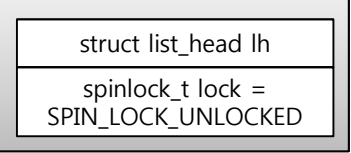


page address pool, page address maps, page address htable 자료구조

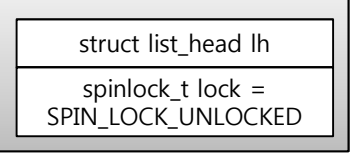


page address htable

page_address_htable[0]

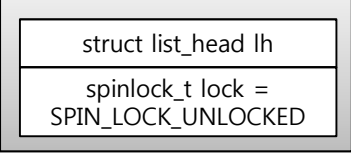


page_address_htable[1]

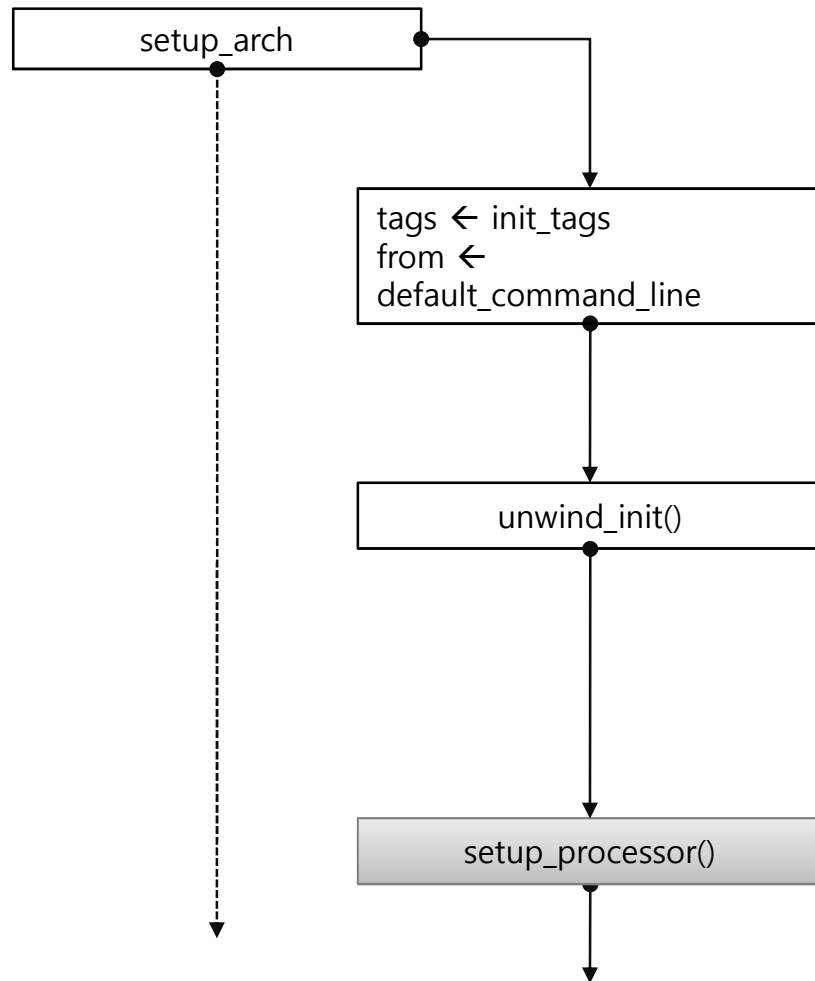


$1 \ll \text{PA_HASH_ORDER} = 0x80 = 128$

page_address_htable[127]



init/main.c



초기 tags값 assign 및 boot시 넘어온 command line assign default command line은 `arch/arm/configs/xxxx_defconfig`에 `CONFIG_CMDLINE`로 정의된 값이며, `__initdata` section에 위치하게 됨.

`CONFIG_ARM_UNWIND`이 정의되었을 경우, 실행되며 정의되지 않은 경우 0을 리턴함.

`.ARM.unwind_idx` section의 처음을 나타내는 `__start_unwind_idx`에서 끝을 나타내는 `__stop_unwind_idx`까지 `unwind_idx` struct 단위로 for 루프를 돌면서 `.ARM.unwind_idx` section에 저장된 symbol 주소를 절대주소로 변경함 symbol 주소는 `unwind_idx->addr`에 저장되어 있음.

arch/arm/include/asm/unwind.h

```
struct unwind_idx {  
    unsigned long addr; /* virtual address 임*/  
    unsigned long insn; /* 현재 instruction 가리킴*/  
};
```

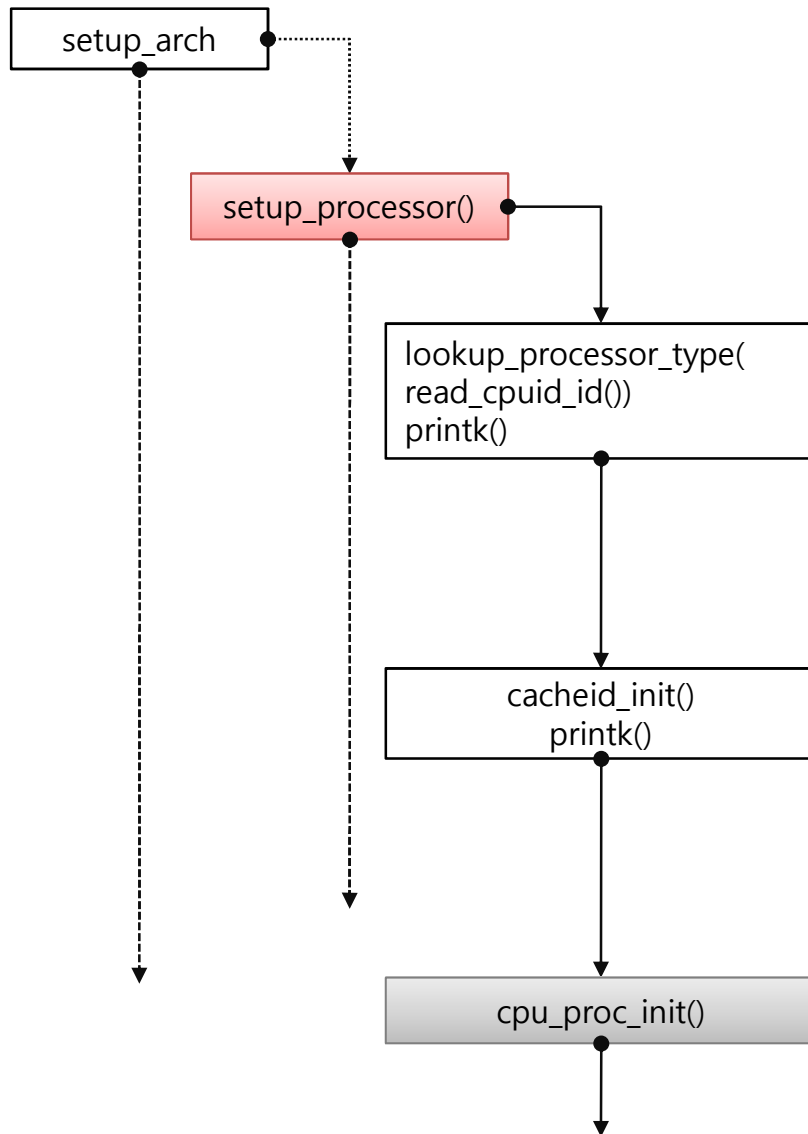
`.ARM.unwind_idx`는 unwinding section에 대한 엔트리를 저장하고 있는 섹션이며 `vmlinux.Ids.S`에서 확인할 수 있고, linker descriptor에 사용된 값은 모두 virtual address값이기 때문에, 실제 물리주소로 변경함.

-g option을 command line에 주게되면, 목적파일은 DWARF2 디버그 프레임 정보를 갖게되고 디버거가 이 정보를 이용하여 필요할 경우 스택을 unwind할 때 이용하게 되어 stack backtrace를 볼 수 있게 됨.

다음문서에서 ELF 포맷 확인할 것

http://infocenter.arm.com/help/topic/com.arm.doc.ih0044d/IHI0044D_aaelf.pdf

init/main.c



arch/kernel/head-common.S에 정의되어 있는 함수로, `proc_info_list`에 processor type을 저장함.

`lookup_processor_type`의 동작방식은 앞장 xxx 참조
`printk("CPU: %s [%08x] revision %d (ARMv%s), cr=%08lxWn",
cpu_name, read_cpuid_id(), read_cpuid_id() & 15,
proc_arch[cpu_architecture()], cr_alignment);`

ex: CPU: ARMv7 Processor [411fc083] revision 3 (ARMv7),
cr=10c5387f

`cpu_architecture()`는 cpuid를 cp15 c0에서 읽어와서 그 값으로 `proc_arch`의 인덱스로 사용하여 architecture를 가져옴.

proc info list 구조체, proc arch 배열 & sample value (v7 proc info)

`read_cpuid_cachetype()`를 통해 cache type을,
`cpu_architecture()`를 통해 arch 정보를 얻어와 cache id를 셋팅함.

cache type에는 4가지가 있음
PIPT (Physically indexed, physically tagged)
VIVT (Virtually indexed, virtually tagged)
VIPT (Virtually indexed, physically tagged)
PIVT (Physically indexed, virtually tagged)

Data cache와 instruction cache가 어떤 type인지 `printk`를 통해 콘솔에 프린트함.

ex: CPU: VIPT nonaliasing data cache, VIPT nonaliasing
instruction cache

proc info list 구조체 & sample value (_v7_proc_info)

arch/arm/mm/proc-v7.S

```
.type __v7_proc_info, #object
__v7_proc_info:
    .long 0x000f0000    @ Required ID value
    .long 0x000f0000    @ Mask for ID
    .long PMD_TYPE_SECT | \
        PMD_SECT_BUFFERABLE | \
        PMD_SECT_CACHEABLE | \
        PMD_SECT_AP_WRITE | \
        PMD_SECT_AP_READ
    .long PMD_TYPE_SECT | \
        PMD_SECT_XN | \
        PMD_SECT_AP_WRITE | \
        PMD_SECT_AP_READ
b __v7_setup
    .long cpu_arch_name
    .long cpu_elf_name
    .long HWCAP_SWP|HWCAP_HALF|HWCAP_THUMB|
        HWCAP_FAST_MULT|HWCAP_EDSP
    .long cpu_v7_name
    .long v7_processor_functions
    .long v7wbi_tlb_fns
    .long v6_user_fns
    .long v7_cache_fns
    .size __v7_proc_info, . - __v7_proc_info
```

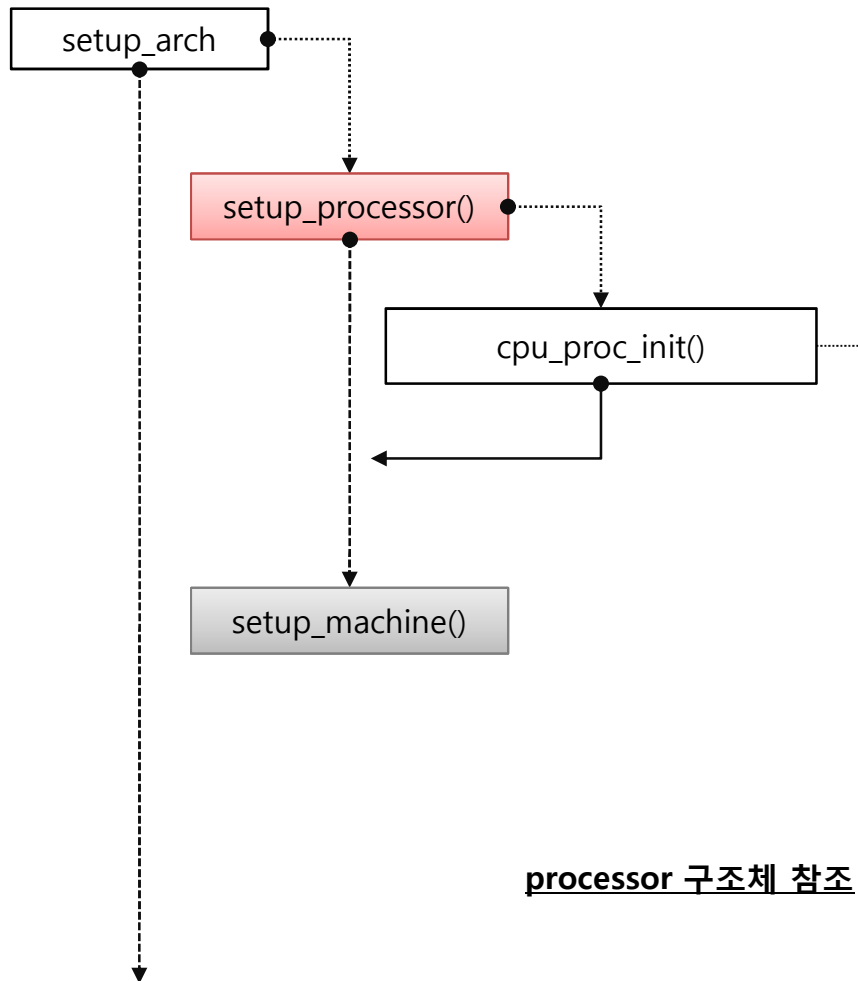
arch/arm/include/asm/procinfo.h

```
struct proc_info_list {
    unsigned int    cpu_val;
    unsigned int    cpu_mask;
    unsigned long   __cpu_mm_mmu_flags; /* used by head.S */
    unsigned long   __cpu_io_mmu_flags; /* used by head.S */
    unsigned long   __cpu_flush;       /* used by head.S */
    const char      *arch_name;
    const char      *elf_name;
    unsigned int    elf_hwcap;
    const char      *cpu_name;
    struct processor *proc;
    struct cpu_tlb_fns *tlb;
    struct cpu_user_fns *user;
    struct cpu_cache_fns *cache;
};
```

arch/arm/kernel/setup.c

```
static const char *proc_arch[] = {
    "undefined/unknown",
    "3", "4", "4T", "5",
    "5T", "5TE", "5TEJ",
    "6TEJ", "7", "? (11)",
    "? (12)", "? (13)", "? (14)",
    "? (15)", "? (16)", "? (17)",
};
```

init/main.c



single cpu일 때와 multi cpu일 때로 나뉘지는데, single cpu일 경우, CPU_NAME과 함수명인 _proc_init를 결합한 함수명을 호출 함. 즉, CPU_NAME = cpu_v7, _proc_init = xxx_proc_init 이면 호출되는 함수는 cpu_v7xxx_proc_init, 임.

multi cpu일 경우, processor 구조체의 _proc_init 함수포인터를 콜, 즉 processor._proc_init()를 콜. processor는 lookup_processor_type()를 통해 결과값을 넘어온 proc_info_list의 멤버이기 때문에, processor의 값들은 proc_info_list.proc으로 액세스가 가능함.

ex: arch/arm/mm/proc-v7.S에 정의된 function list

```
.type v7_processor_functions, #object
ENTRY(v7_processor_functions)
.word v7_early_abort
.word pabort_ifar
.word cpu_v7_proc_init
.word cpu_v7_proc_fin
.word cpu_v7_reset
.word cpu_v7_do_idle
.word cpu_v7_dcache_clean_area
.word cpu_v7_switch_mm
.word cpu_v7_set_pte_ext
.size v7_processor_functions, . - v7_processor_functions
```

이 경우, processor._proc_init = cpu_v7_proc_init 이고 그 정의

```
ENTRY(cpu_v7_proc_init)
    mov pc, lr
ENDPROC(cpu_v7_proc_init)
```

proc info list 구조체 & sample value (_v7_proc_info)

arch/arm/include/asm/cpu-multi32.h

```
extern struct processor {
    /* MISC: get data abort address/flags */
    void (*_data_abort)(unsigned long pc);

    unsigned long (*_prefetch_abort)(unsigned long lr); /* Retrieve prefetch fault address */

    void (*_proc_init)(void); /* Set up any processor specifics */

    void (*_proc_fin)(void); /* Disable any processor specifics */

    void (*reset)(unsigned long addr) __attribute__((noreturn)); /* Special stuff for a reset */

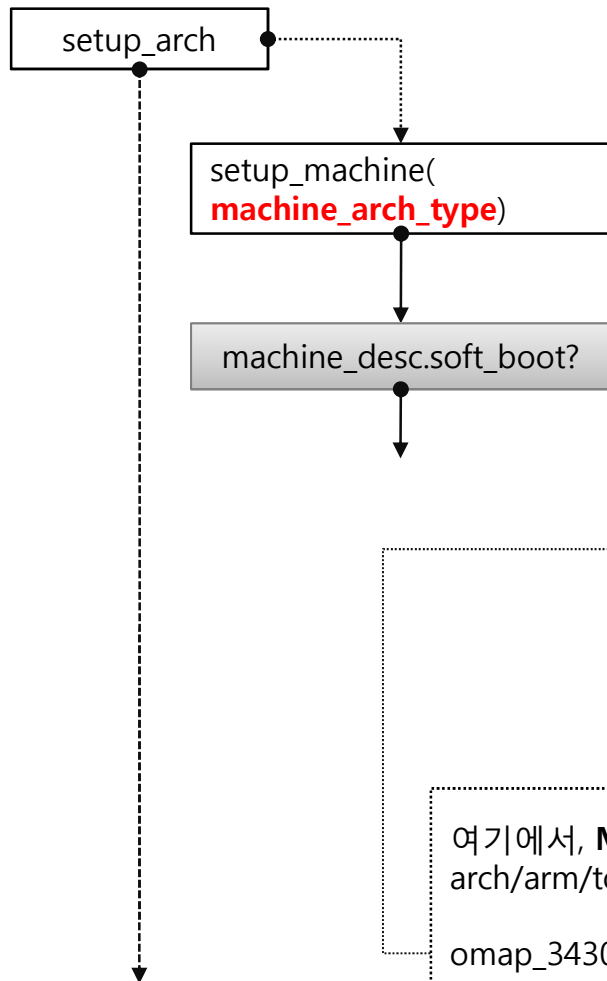
    int (*_do_idle)(void); /* Idle the processor */

    /* Processor architecture specific */
    /* clean a virtual address range from the D-cache without flushing the cache. */
    void (*dcache_clean_area)(void *addr, int size);

    void (*switch_mm)(unsigned long pgd_phys, struct mm_struct *mm); /* Set the page table */

    /* Set a possibly extended PTE. Non-extended PTEs should ignore 'ext'. */
    void (*set_pte_ext)(pte_t *ptep, pte_t pte, unsigned int ext);
} processor;
```


init/main.c



arch/arm/tools/gen-mach-types의 awk script이 include/asm-arm/mach-types.h 파일을 생성하는데, mach-types.h에 machine_arch_type에 대한 정의를 함.

#define machine_arch_type __machine_arch_type

machine_arch_type은 head-common.S의 __lookup_machine_type에 의해 machine_arch_type이 해당 machine descriptor의 메모리상의 위치를 가르키고, 그 첫 번째 멤버인 architecture number가 됨, 즉 machine_desc.nr를 가리킴.

ex: arch/arm/mach-omap2/board-3430sdp.c

MACHINE_START(OMAP_3430SDP, "OMAP3430 3430SDP board")

/* Maintainer: Syed Khasim - Texas Instruments Inc */

.phys_io = 0x48000000,

.io_pg_offst = ((0xd8000000) >> 18) & 0xfffc,

.boot_params = 0x80000100,

.map_io = omap_3430sdp_map_io,

.init_irq = omap_3430sdp_init_irq,

.init_machine = omap_3430sdp_init,

.timer = &omap_timer,

MACHINE_END

여기에서, **MACHINE_START**매크로에 의해 .nr = MACH_TYPE_OMAP_3430SDP_type이 되며, arch/arm/tools/mach-types에 아래와 같이 정의되어 있음.

omap_3430sdp	MACH_OMAP_3430SDP	OMAP_3430SDP	1138
--------------	-------------------	--------------	------

콘솔에 machine_desc.name을 다음과 같이 출력, printk("Machine: %s\n", list->name)

ex: Machine: My Company Board

machine desc 구조체 / MACHINE START 매크로참조

machine_desc 구조체 / MACHINE_START 매크로

arch/arm/include/asm/mach/arch.h

```
struct machine_desc {
    /*
     * Note! The first four elements are used
     * by assembler code in head.S, head-common.S
     */
    unsigned int    nr;    /* architecture number */
    unsigned int    phys_io; /* start of physical io */
    unsigned int    io_pg_offst; /* byte offset for io
     * page table entry */

    const char    *name;    /* architecture name */
    unsigned long    boot_params; /* tagged list */

    unsigned int    video_start; /* start of video RAM */
    unsigned int    video_end; /* end of video RAM */
    unsigned int    reserve_lp0 :1; /* never has lp0 */
    unsigned int    reserve_lp1 :1; /* never has lp1 */
    unsigned int    reserve_lp2 :1; /* never has lp2 */
    unsigned int    soft_reboot :1; /* soft reboot */

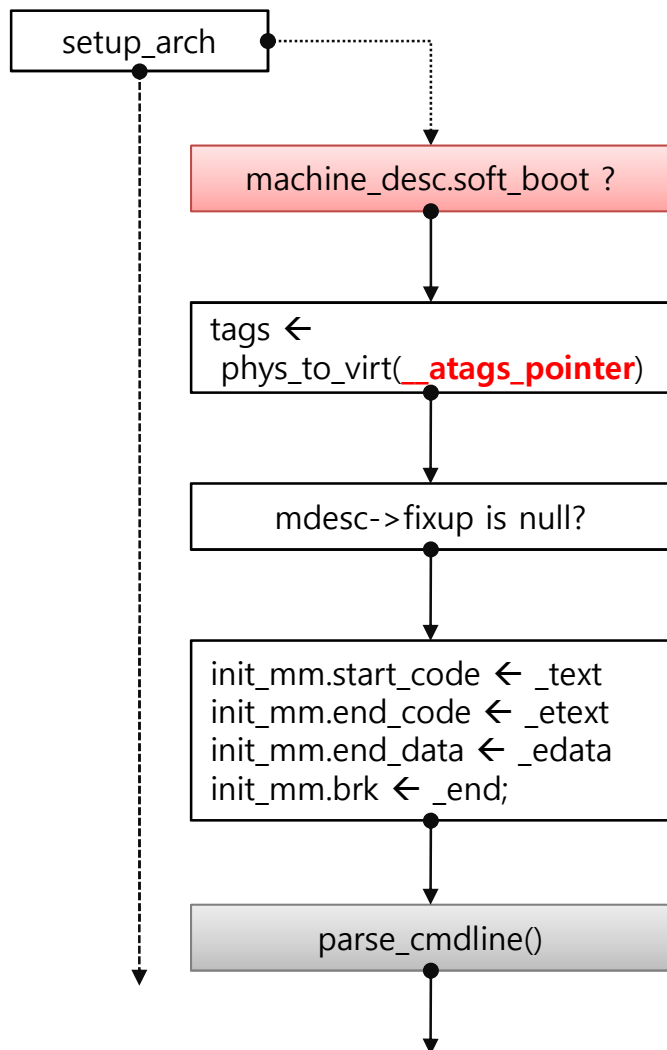
    void    (*fixup)(struct machine_desc *,
        struct tag *, char **,
        struct meminfo *);
    void    (*map_io)(void); /* IO mapping function */
    void    (*init_irq)(void);
    struct sys_timer    *timer; /* system tick timer */
    void    (*init_machine)(void);
};
```

arch/arm/include/asm/mach/arch.h

```
#define MACHINE_START(_type, _name) \
static const struct machine_desc __mach_desc_##_type \
__used \
__attribute__((__section__(".arch.info.init"))) = { \
    .nr    = MACH_TYPE_##_type, \
    .name    = _name,

#define MACHINE_END \
};
```

init/main.c



만약 `machine_desc.soft_boot`가 정의되어 있다면, `reboot_mode = "s"`로 assign.하는 `reboot_setup("s")` 콜. Soft reboot????

`__atags_pointer`는 `arch/arm/kernel/head-common.S`에서 부트로더에서 넘어온 `atags` 주소 를 가르키고 있다. `__atags_pointer`를 가상주소로 변경하여 `tags`에 assign. `arch/arm/kernel/setup.c`에 다음과 같이 정의됨

```
unsigned int __atags_pointer __initdata;
```

만일 `__atags_pointer`가 null이라면, `machine_desc.boot_params`를 가상주소로 변경함. 만일 `machine_desc.boot_params`까지 null이라면, `init_tags`의 값을 이용함.

`mdesc->fixup`이 null이 아닐 경우, `mdesc->fixup` 수행하며, 이 함수는 메모리 관련정보를 설정한다. 메모리 관련정보의 설정은 아래 3곳 중 한곳에서 한다

1. 머신의 `fixup`함수에서 `meminfo`를 전달받아서 설정
2. 부트로더에서 `ATAG_MEM`태그를 전달해서 설정
3. 커널 커맨드라인에서 `mem=xxx` 파라미터를 전달해서 설정

Init task의 `mm_struct`의 `_text`, `_etext`, `_edata`, `_end` 위치를 assign하며, `arch/alpha/kernel/init_task.c`에 `init_mm`은 정적으로 생성된다.

```
struct mm_struct init_mm = INIT_MM(init_mm);
```

mm struct 구조체 & INIT MM 매크로

mm_struct 구조체 & INIT MM 매크로

include/linux/init_task.h

include/linux/mm_types.h

```
struct mm_struct {
    struct vm_area_struct * mmap; /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    unsigned long (*get_unmapped_area) (struct file *filp,
        unsigned long addr, unsigned long len,
        unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
    unsigned long mmap_base; /* base of mmap area */
    unsigned long task_size; /* size of task vm space */

    /* if non-zero, the largest hole below free_area_cache */
    unsigned long cached_hole_size;
    unsigned long free_area_cache; /* first hole of size cached_hole_size or larger */
    pgd_t * pgd;
    atomic_t mm_users; /* How many users with user space? */
    atomic_t mm_count; /* How many references to "struct mm_struct" (users count as 1) */
    int map_count; /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock; /* Protects page tables and some counters */

    struct list_head mmlist;

    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;

    unsigned long hiwater_rss; /* High-watermark of RSS usage */
    unsigned long hiwater_vm; /* High-water virtual memory usage */

    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
```

```
#define INIT_MM(name) \
{ \
    .mm_rb = RB_ROOT, \
    .pgd = swapper_pg_dir, \
    .mm_users = ATOMIC_INIT(2), \
    .mm_count = ATOMIC_INIT(1), \
    .mmap_sem = __RWSEM_INITIALIZER(name.mmap_sem), \
    .page_table_lock = __SPIN_LOCK_UNLOCKED(name.page_table_lock), \
    .mmlist = LIST_HEAD_INIT(name.mmlist), \
    .cpu_vm_mask = CPU_MASK_ALL, \
}
```

include/linux/mm_types.h (continue)

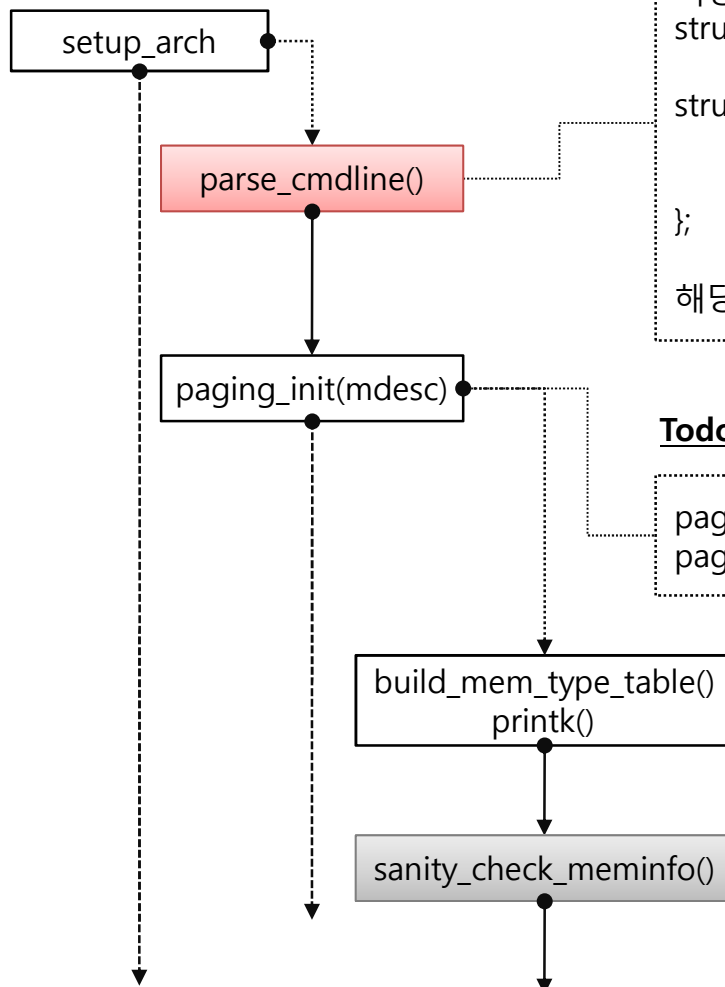
```
/* for /proc/PID/auxv */
unsigned long saved_auxv[AT_VECTOR_SIZE];
cpumask_t cpu_vm_mask;

/* Architecture-specific MM context */
mm_context_t context;

unsigned int faultstamp;
unsigned int token_priority;
unsigned int last_interval;
unsigned long flags; /* Must use atomic bitops to access the bits */
struct core_state *core_state; /* coredumping support */

/* aio bits */
spinlock_t ioctx_lock;
struct hlist_head ioctx_list;
};
```

init/main.c



부트로더에서 넘어온 command line을 파싱한다. command line은 .init section에 저장되어 있고, `__early_begin`를 시작으로 `__early_end`까지, `struct early_params` 단위로 액세스하며, 공백이 delimiter로 이용됨.

```
struct early_params {  
    const char *arg;  
    void (*fn)(char **p);  
};
```

해당 arg (ex: console=ttySAC0,115200)에 대해 정의된 `early_params.fn`을 호출함.

Todo: 각 arg에 대해 정의된 함수는 어디에 있으며, 실체를 파악하자

page table들을 셋업하고, zone memory map들을 초기화하며, zero page, bad page, bad page 테이블을 만든다.

정적 `mem_types` 테이블은 memory type에 대한 entry들로 구성되며 각 엔트리는 아키텍처를 검사하여 필요없는 비트는 클리어시키고 필요한 비트는 세트 시킨다.

cache policy를 설정함. `CONFIG_SMP`가 정의된 경우 cache policy는 `CPOLICY_WRITEALLOC`로 셋팅 됨

Memory policy에 대한 `printk`를 수행함.
ex: Memory policy: ECC disabled, Data cache writeback

Cache policy & 정적 mem types 구조체 배열에 대한 정리

mem_type 구조체와 mem_types 정적 구조체 배열

```
static struct mem_type mem_types[] = {
    [MT_DEVICE] = {
        /* Strongly ordered / ARMv6 shared device */
        .prot_pte = PROT_PTE_DEVICE |
            L_PTE_MT_DEV_SHARED |
            L_PTE_SHARED,
        .prot_l1 = PMD_TYPE_TABLE,
        .prot_sect = PROT_SECT_DEVICE |
            PMD_SECT_S,
        .domain = DOMAIN_IO,
    },
    [MT_DEVICE_NONSHARED] = {
        /* ARMv6 non-shared device */
        .prot_pte = PROT_PTE_DEVICE |
```

arch/arm/mm/mm.h

```
struct mem_type {
    unsigned int prot_pte;
    unsigned int prot_l1;
    unsigned int prot_sect;
    unsigned int domain;
};
```

```
L_PTE_MT_DEV_NONSHARED,
    .prot_l1 = PMD_TYPE_TABLE,
    .prot_sect = PROT_SECT_DEVICE,
    .domain = DOMAIN_IO,
},
    [MT_DEVICE_CACHED] = {
        /* ioremap_cached */
        .prot_pte = PROT_PTE_DEVICE |
            L_PTE_MT_DEV_CACHED,
        .prot_l1 = PMD_TYPE_TABLE,
        .prot_sect = PROT_SECT_DEVICE |
            PMD_SECT_WB,
        .domain = DOMAIN_IO,
    },
    [MT_DEVICE_WC] = { /* ioremap_wc */
        .prot_pte = PROT_PTE_DEVICE |
            L_PTE_MT_DEV_WC,
        .prot_l1 = PMD_TYPE_TABLE,
        .prot_sect = PROT_SECT_DEVICE,
        .domain = DOMAIN_IO,
    },
    [MT_UNCACHED] = {
        .prot_pte = PROT_PTE_DEVICE,
        .prot_l1 = PMD_TYPE_TABLE,
        .prot_sect = PMD_TYPE_SECT |
            PMD_SECT_XN,
        .domain = DOMAIN_IO,
    },
    [MT_CACHECLEAN] = {
        .prot_sect = PMD_TYPE_SECT |
            PMD_SECT_XN,
        .domain = DOMAIN_KERNEL,
    },
},
```

arch/arm/mm/mmu.c

```
[MT_LOW_VECTORS] = {
    .prot_pte = L_PTE_PRESENT |
        L_PTE_YOUNG |
        L_PTE_DIRTY |
        L_PTE_EXEC,
    .prot_l1 = PMD_TYPE_TABLE,
    .domain = DOMAIN_USER,
},
    [MT_HIGH_VECTORS] = {
        .prot_pte = L_PTE_PRESENT |
            L_PTE_YOUNG |
            L_PTE_DIRTY |
            L_PTE_USER |
            L_PTE_EXEC,
        .prot_l1 = PMD_TYPE_TABLE,
        .domain = DOMAIN_USER,
    },
    [MT_MEMORY] = {
        .prot_sect = PMD_TYPE_SECT |
            PMD_SECT_AP_WRITE,
        .domain = DOMAIN_KERNEL,
    },
    [MT_ROM] = {
        .prot_sect = PMD_TYPE_SECT,
        .domain = DOMAIN_KERNEL,
    },
    [MT_MEMORY_NONCACHED] = {
        .prot_sect = PMD_TYPE_SECT |
            PMD_SECT_AP_WRITE,
        .domain = DOMAIN_KERNEL,
    },
};
```

cachepolicy 구조체

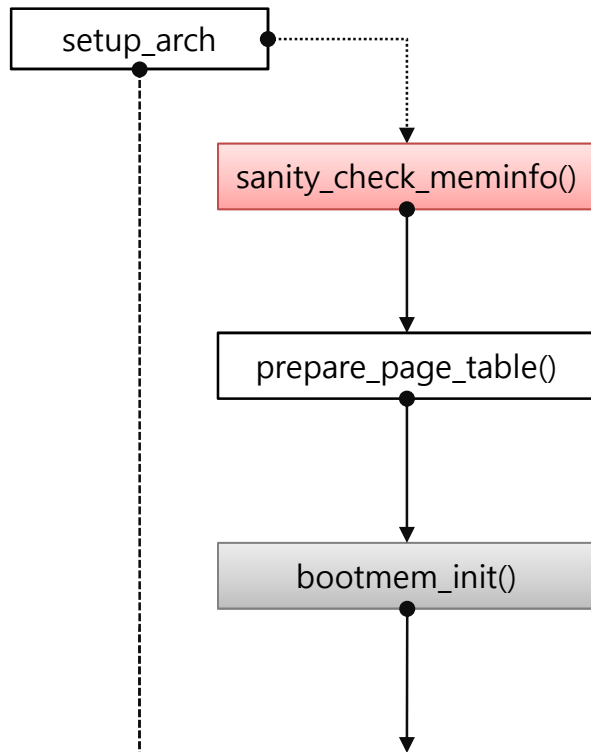
arch/arm/mm/mmu.c

```
struct cachepolicy {  
    const char  policy[16];  
    unsigned int cr_mask;  
    unsigned int pmd;  
    unsigned int pte;  
};
```

arch/arm/mm/mmu.c

```
static struct cachepolicy cache_policies[] __initdata = {  
    {  
        .policy    = "uncached",  
        .cr_mask   = CR_W|CR_C,  
        .pmd       = PMD_SECT_UNCACHED,  
        .pte       = L_PTE_MT_UNCACHED,  
    }, {  
        .policy    = "buffered",  
        .cr_mask   = CR_C,  
        .pmd       = PMD_SECT_BUFFERED,  
        .pte       = L_PTE_MT_BUFFERABLE,  
    }, {  
        .policy    = "writethrough",  
        .cr_mask   = 0,  
        .pmd       = PMD_SECT_WT,  
        .pte       = L_PTE_MT_WRITETHROUGH,  
    }, {  
        .policy    = "writeback",  
        .cr_mask   = 0,  
        .pmd       = PMD_SECT_WB,  
        .pte       = L_PTE_MT_WRITEBACK,  
    }, {  
        .policy    = "writealloc",  
        .cr_mask   = 0,  
        .pmd       = PMD_SECT_WBWA,  
        .pte       = L_PTE_MT_WRITEALLOC,  
    }  
};
```

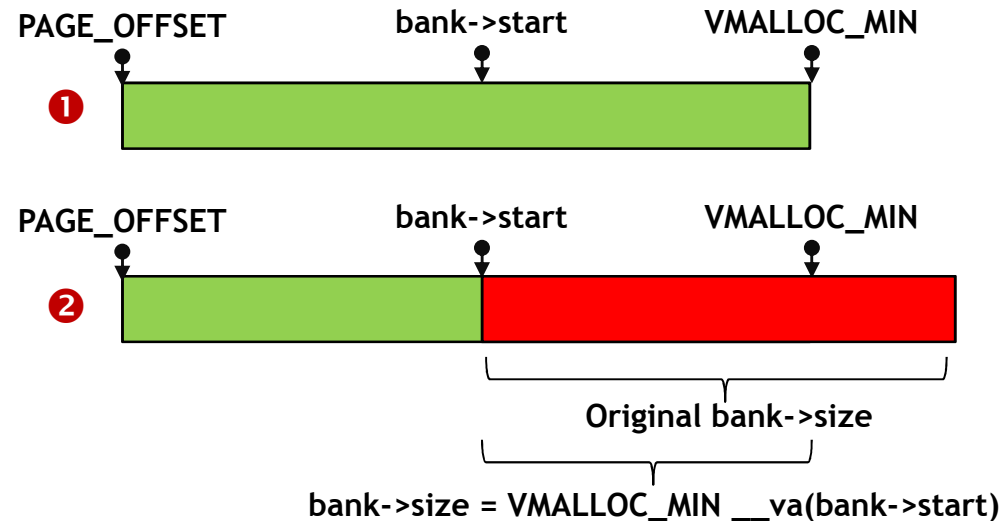
init/main.c



A fixup table -- a relocation table -- can also be provided in the header of the object code file. Each "fixup" is a pointer to an address in the object code that must be changed when the loader relocates the program.

예로 mach-pxa의 machine descriptor에 정의된 pixup 함수에 의해 meminfo가 채워지고, 여기에서 sanity 검사를 함. 각 bank에 대해서 bank->size가 VMALLOC_MIN을 넘어가는 영역을 빼고 다시 resize를 시킴.

$VMALLOC_MIN = VMALLOC_END - vmalloc_reserve$ (default 128MB)
 $bank->size = VMALLOC_MIN - _va(bank->start)$

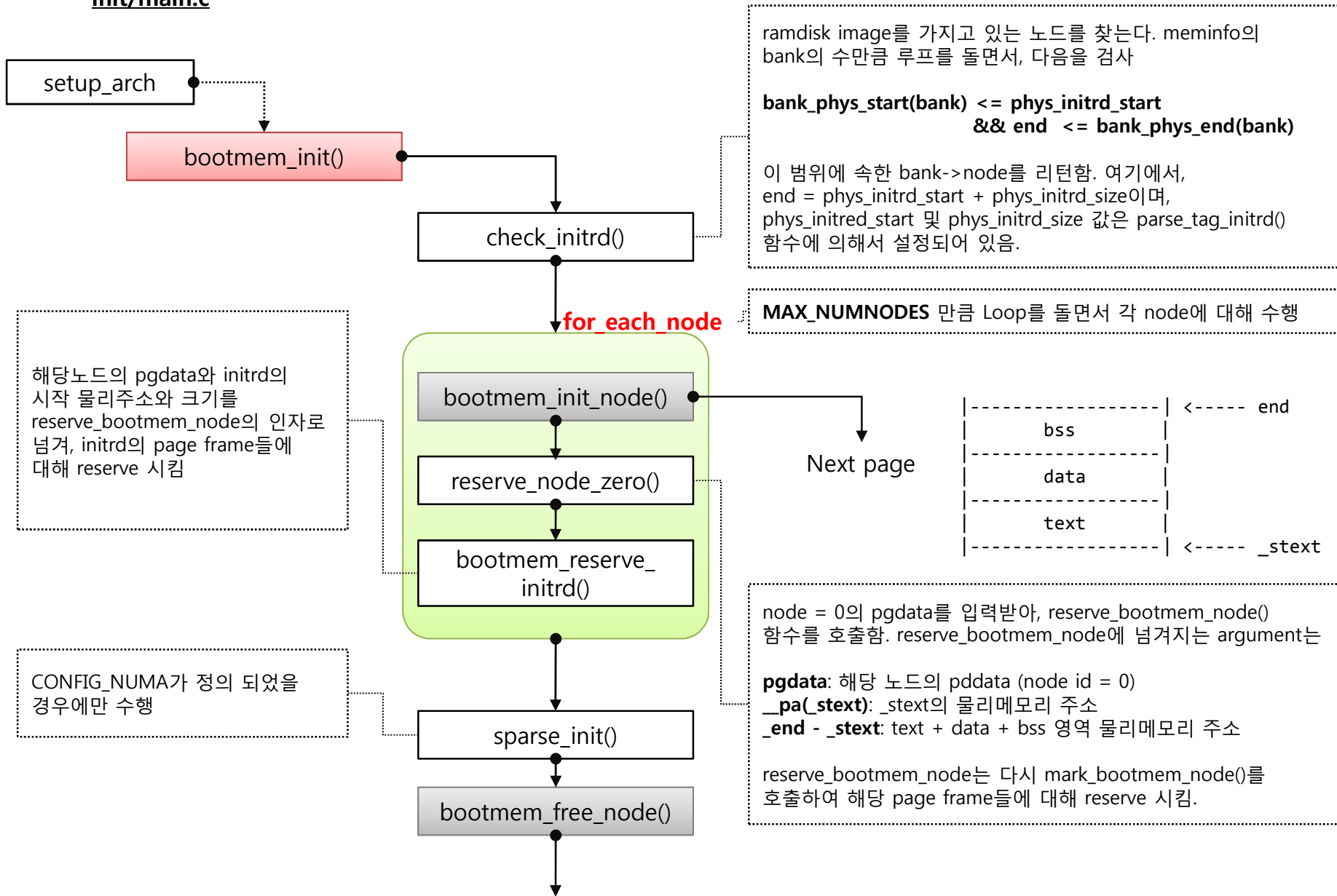


pgd page table을 초기화하는데, 크게 다음단계로 수행됨

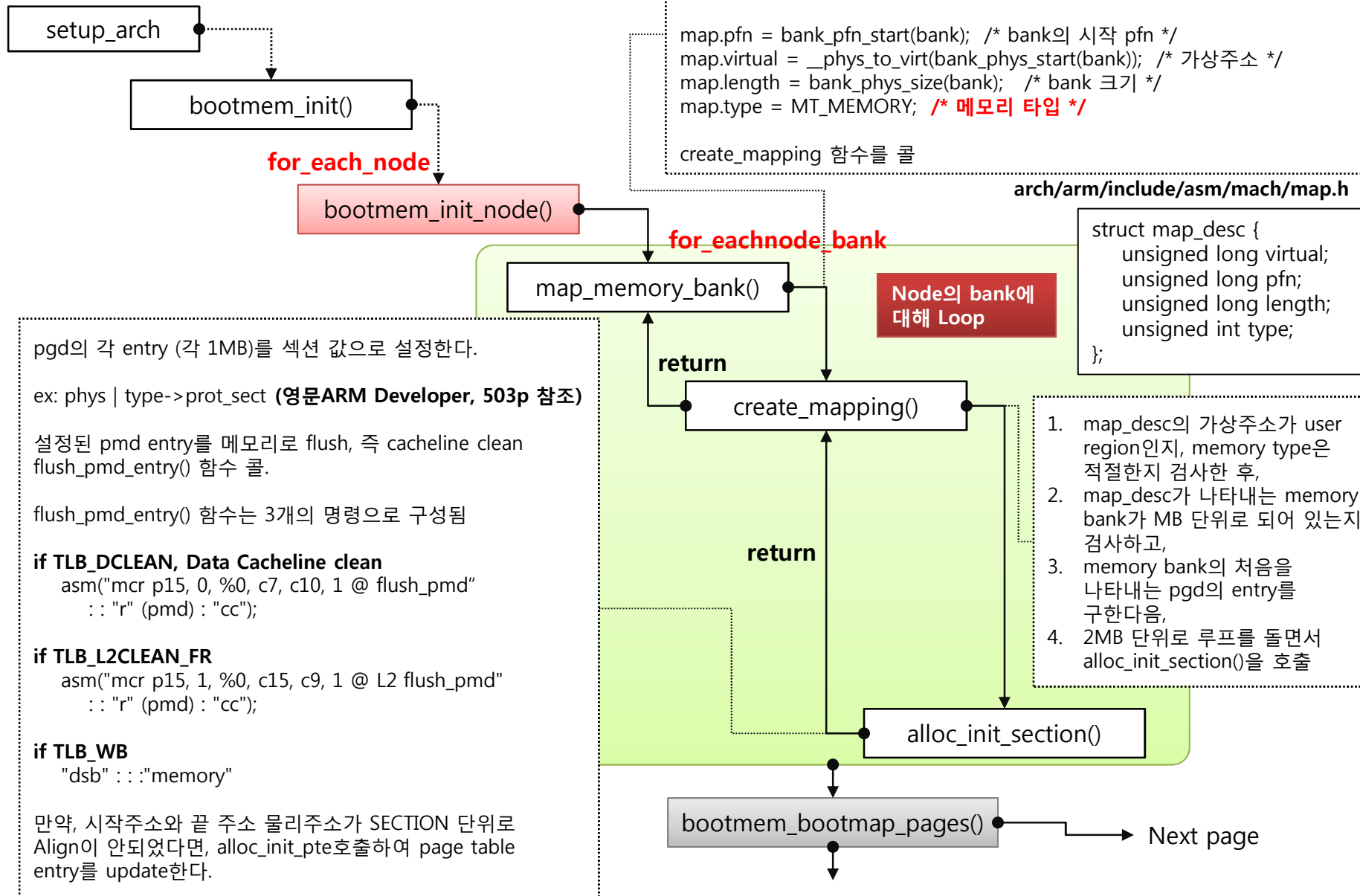
1. virtual address = 0에서 `PAGE_OFFSET - 16MB` 까지
2. `CONFIG_XIP_KERNEL`이 정의되었을 경우, 1MB로 align하여 `_etext + 1MB` 이후로 초기화할 address를 지정
3. `CONFIG_XIP_KERNEL`가 정의되지 않았을 경우, `PAGE_OFFSET`까지 pgd table entry를 초기화
4. `bank[0]`의 size만큼을 제외한후 `VMALLOC_END`까지 pgd table entry를 초기화

todo: memory bank, node, vmalloc area에 대해서 정리하기

init/main.c



init/main.c



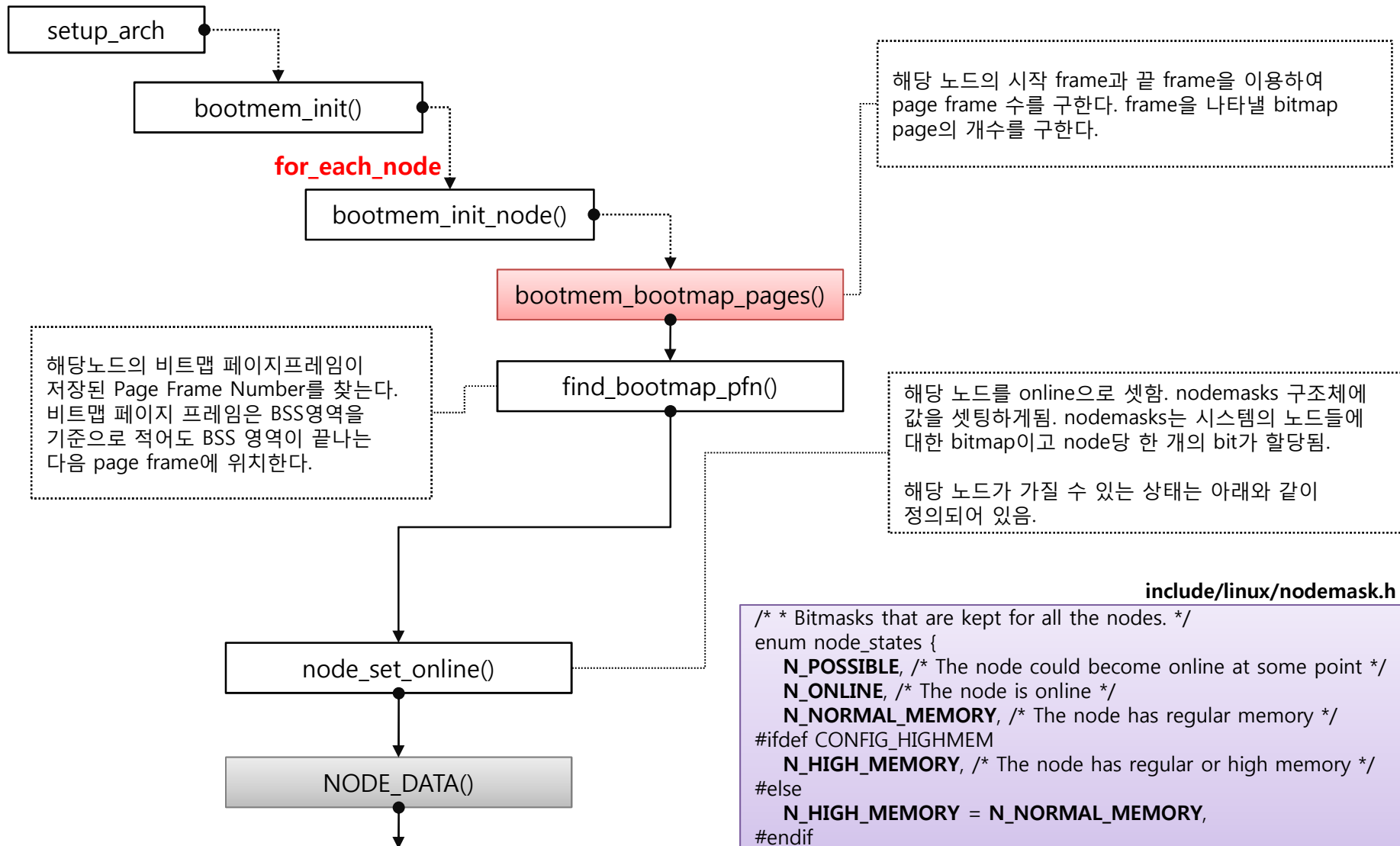
Memory Types

arch/arm/include/asm/mach/map.h /* types 0-3 are defined in asm/io.h */

```
#define MT_UNCACHED 4
#define MT_CACHECLEAN 5
#define MT_MINICLEAN 6
#define MT_LOW_VECTORS 7
#define MT_HIGH_VECTORS 8
#define MT_MEMORY 9
#define MT_ROM 10
#define MT_MEMORY_NONCACHED 11
```

Todo: 정리

init/main.c

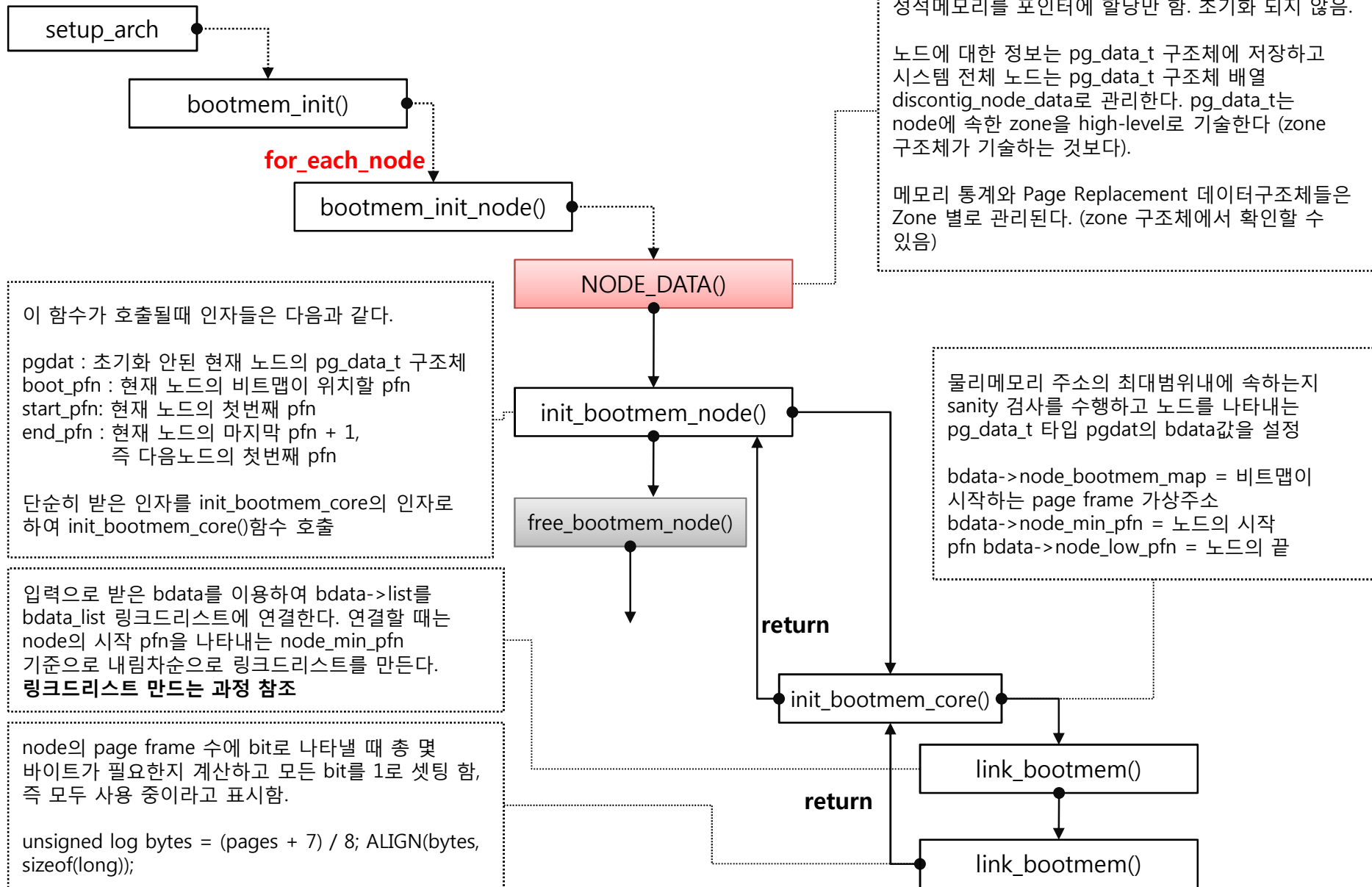


include/linux/nodemask.h

```

/* * Bitmasks that are kept for all the nodes. */
enum node_states {
    N_POSSIBLE, /* The node could become online at some point */
    N_ONLINE, /* The node is online */
    N_NORMAL_MEMORY, /* The node has regular memory */
#ifdef CONFIG_HIGHMEM
    N_HIGH_MEMORY, /* The node has regular or high memory */
#else
    N_HIGH_MEMORY = N_NORMAL_MEMORY,
#endif
    N_CPU, /* The node has one or more cpus */
    NR_NODE_STATES
};
  
```

init/main.c

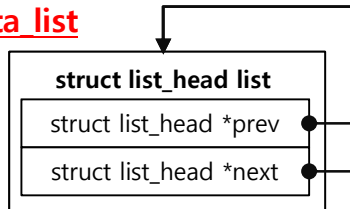


Todo: bdata->list 링크드 리스트

bdata->list 링크드 리스트

①

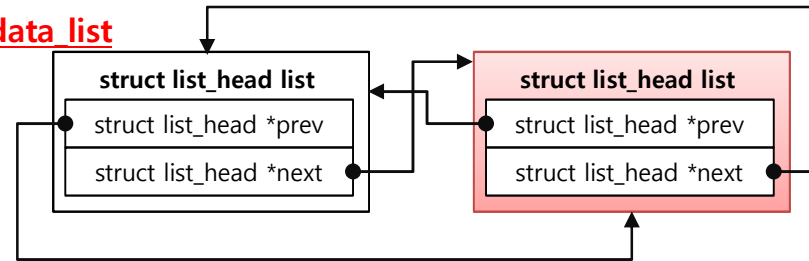
bdata list



Initial State

②

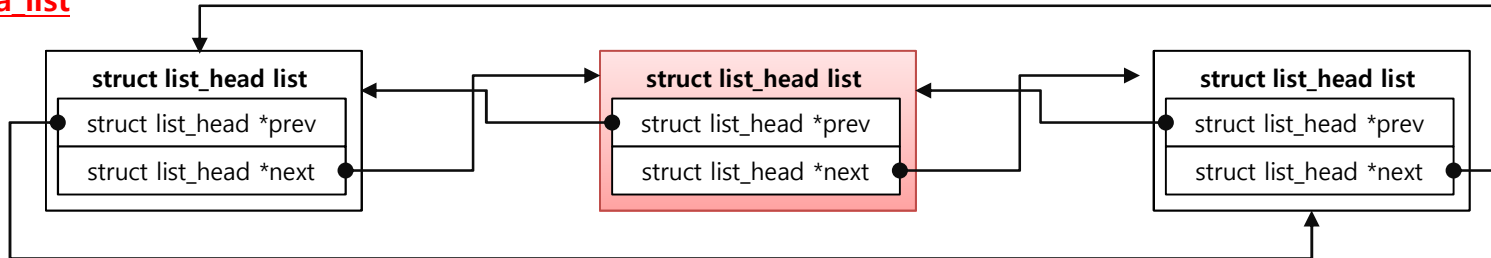
bdata list



list_add_tail(&bdata->list, iter) 호출 후

③

bdata list

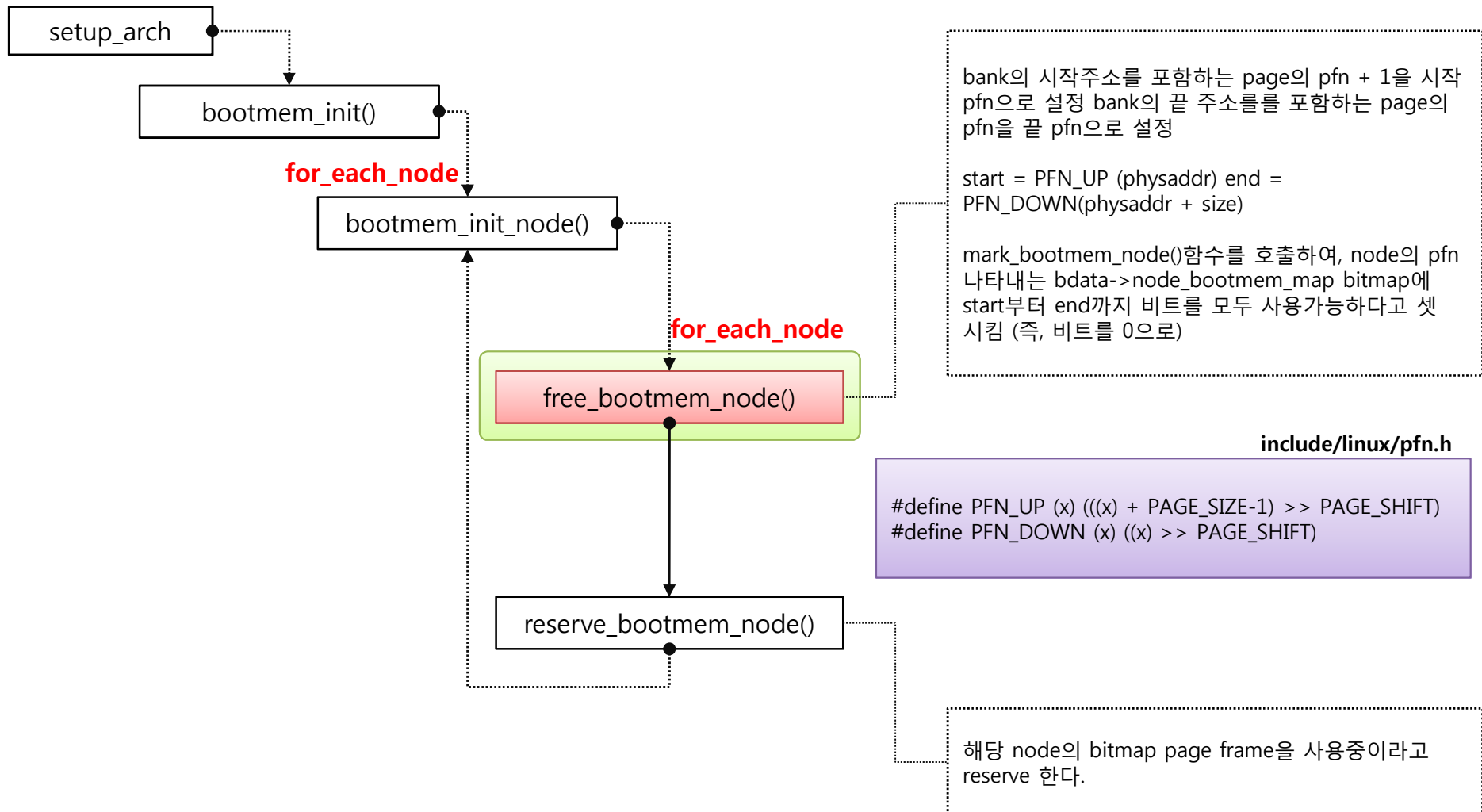


중간 item을 찾고 list_add_tail(&bdata->list, iter) 호출 후

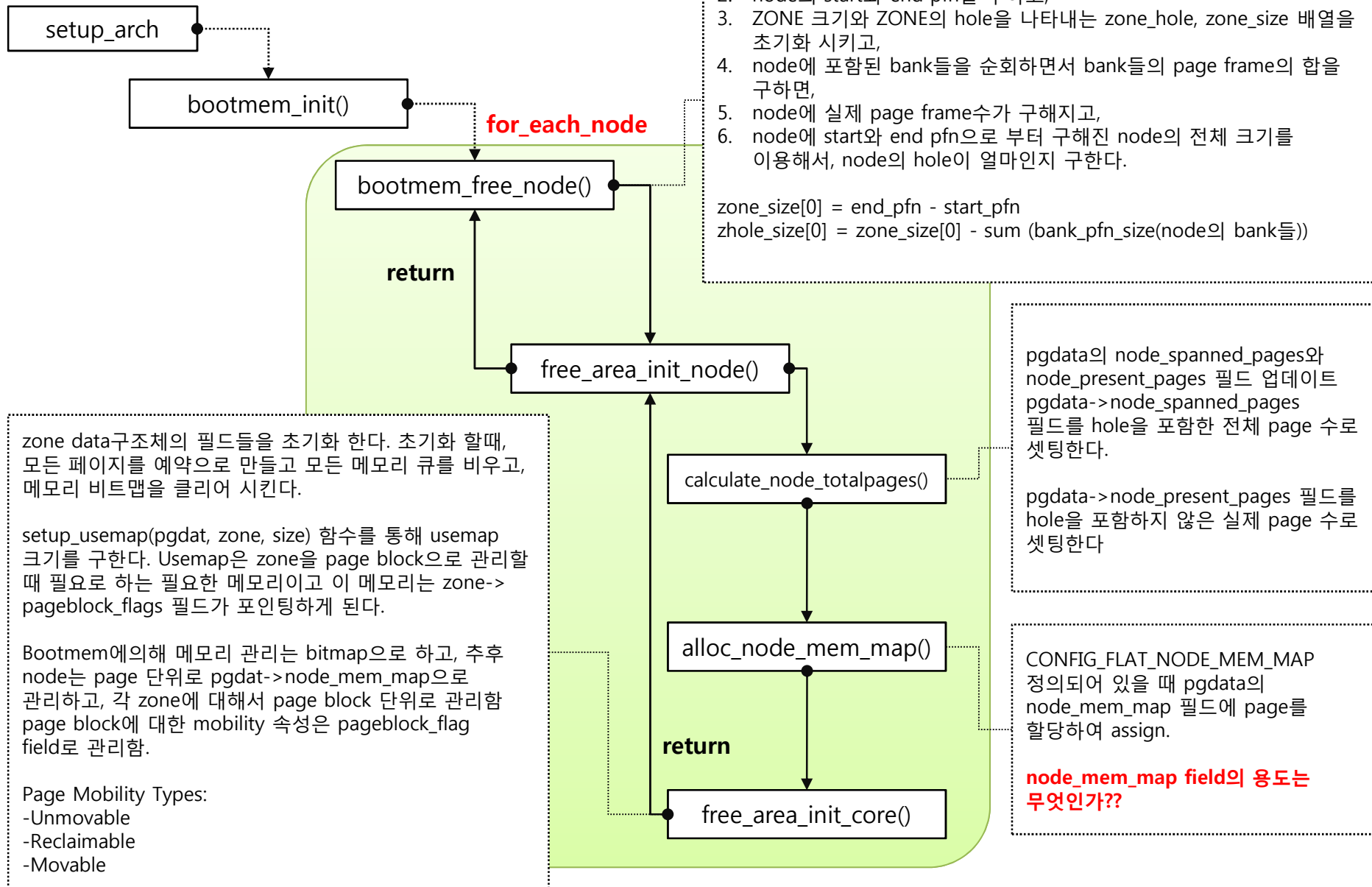
```
typedef struct bootmem_data {
    unsigned long node_min_pfn;
    unsigned long node_low_pfn;
    void *node_bootmem_map;
    unsigned long last_end_off;
    unsigned long hint_idx;
    struct list_head list;
} bootmem_data_t;

/* 노드의 시작 pfn */
/* 노드의 마지막 pfn */
/* 관리용 비트맵 */
/* 마지막으로 할당한 위치 */
/* 다음에 사용될 페이지(힌트) */
/* 다른 노드의 bootmem_data를 연결하는 포인터 */
```

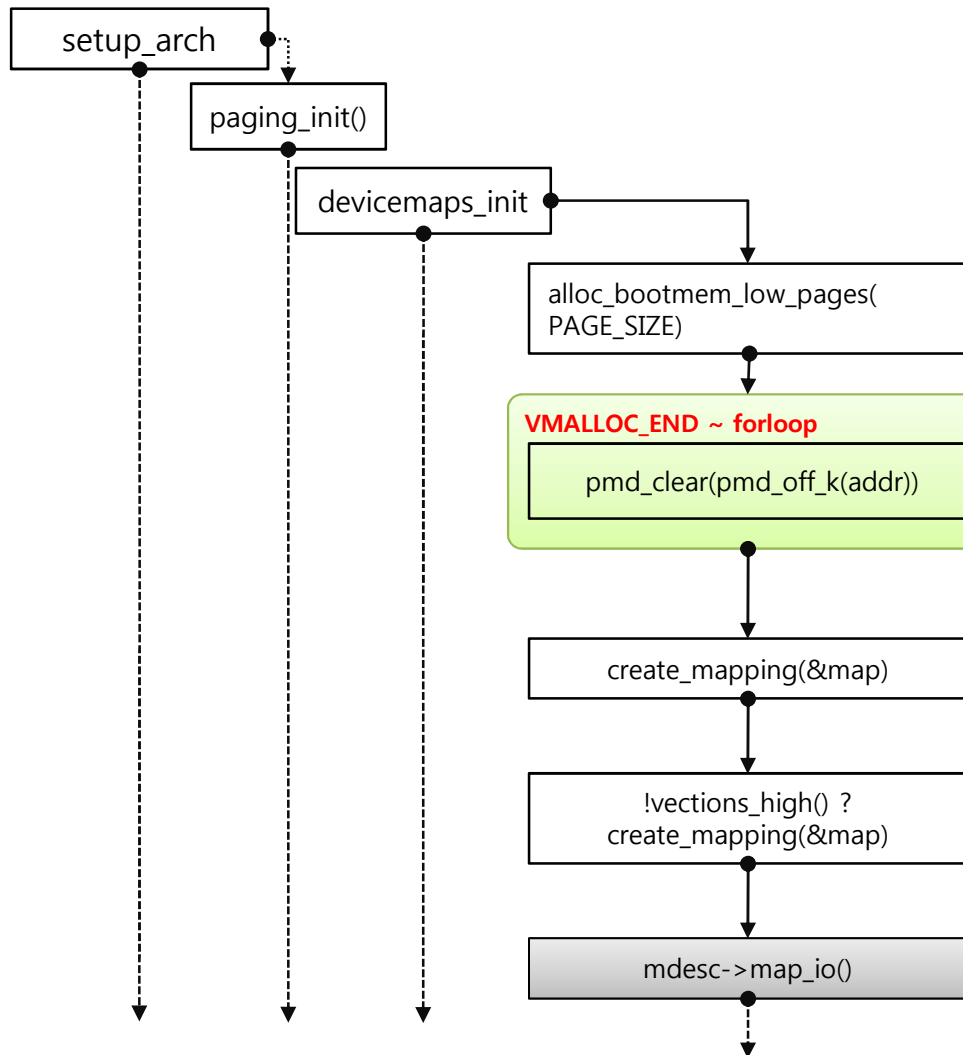
init/main.c



init/main.c



init/main.c



PAGE_SIZE크기만큼의 메모리를 할당해 시작주소를 리턴

Architecture에서 ARCH_LOW_ADDRESS_LIMIT이 정해져있으면 해당 Limit 주소 아래에서 메모리를 할당하고, 정의가 되지 않았다면, 0xffffffffUL 이용

./mm/bootmem.c:983:

#ifndef ARCH_LOW_ADDRESS_LIMIT

./mm/bootmem.c:984:

#define ARCH_LOW_ADDRESS_LIMIT 0xffffffffUL

PGDIR_SIZE = 1UL << 21

2MB 단위로

VMALLOC_END보다 상위주소를 가리키는 pgd엔트리를 모두 초기화시킴. 커널부팅 초기에 pgd를 만들고 커널코드영역만 매핑을 수행했었음. devicemap_init에서는 VMALLOC_END위의 영역을 초기화하고 devicemap을 매핑하는것처럼 보임.

high-vectors에 해당하는 mapping 구조체의 값을 채워 pgd entry & pte entry 값을 update 함. 0xffff0000 주소를 vectors의 물리주소로 매핑

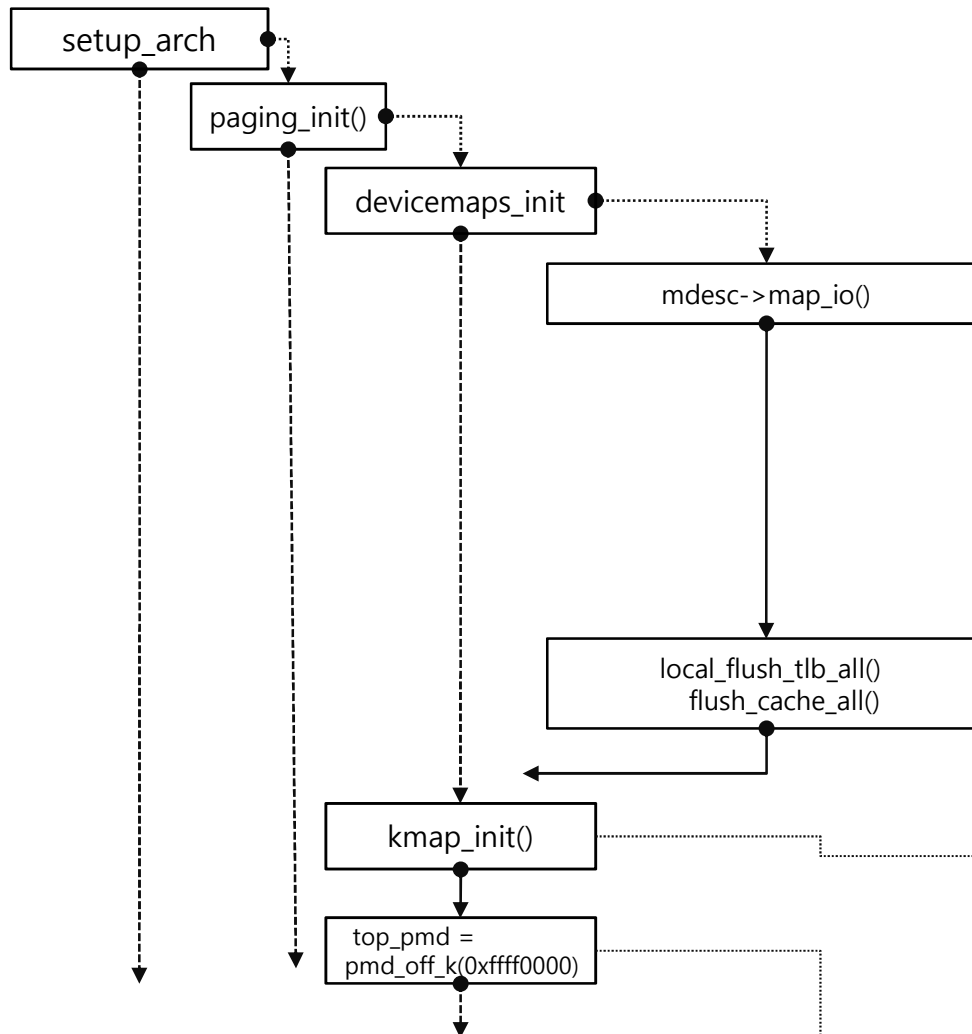
주의: map.length가 1MB보다 작기 때문에, create_mapping()->alloc_init_section() 함수에서 pte table을 생성하는 로직으로 분기됨.

vectors_high가 아니라면, virtual address를 0으로 하고 메모리타입을 MT_LOW_VECTORS로 mapping 구조체의 값을 채워서 create_mapping() 함수를 호출하여 pgd entry & pte entry 값을 update 함.

위쪽을 실행하기전에 vectors_high()를 호출하여 분기하는 것이 좋을 것같은데, 왜 이렇게 하는지...

0xffff0000, 0x00000000 에 해당하는 pte entry들이 동일한 page를 가르키고 있는 셈.

init/main.c



io 장치를 pgd/ pte entry에 매핑
io mapping function인 mdesc->map_io가 NULL이 아니면,
mdesc->map_io()함수를 호출

machine별로 정의되어 있고 예제는 아래 참조
example: ./arm/arm/mach-s3c6410/mach-smdk6410.c

```

MACHINE_START(SMDK6410, "SMDK6410")
/* Maintainer: Ben Dooks <ben@fluff.org> */
.phys_io   = S3C_PA_UART & 0xffff0000,
.io_pg_offst = (((u32)S3C_VA_UART) >> 18) & 0xfffc,
.boot_params = S3C64XX_PA_SDRAM + 0x100,
.init_irq   = s3c6410_init_irq,
.map_io     = smdk6410_map_io,
.init_machine = smdk6410_machine_init,
.timer     = &s3c24xx_timer,
MACHINE_END

static void __init smdk6410_map_io(void)
{
    s3c64xx_init_io(smdk6410_iodesc, ARRAY_SIZE(smdk6410_iodesc));
    s3c24xx_init_clocks(12000000);
    s3c24xx_init_uarts(smdk6410_uartcfgs, ARRAY_SIZE(smdk6410_uartcfgs));
}
    
```

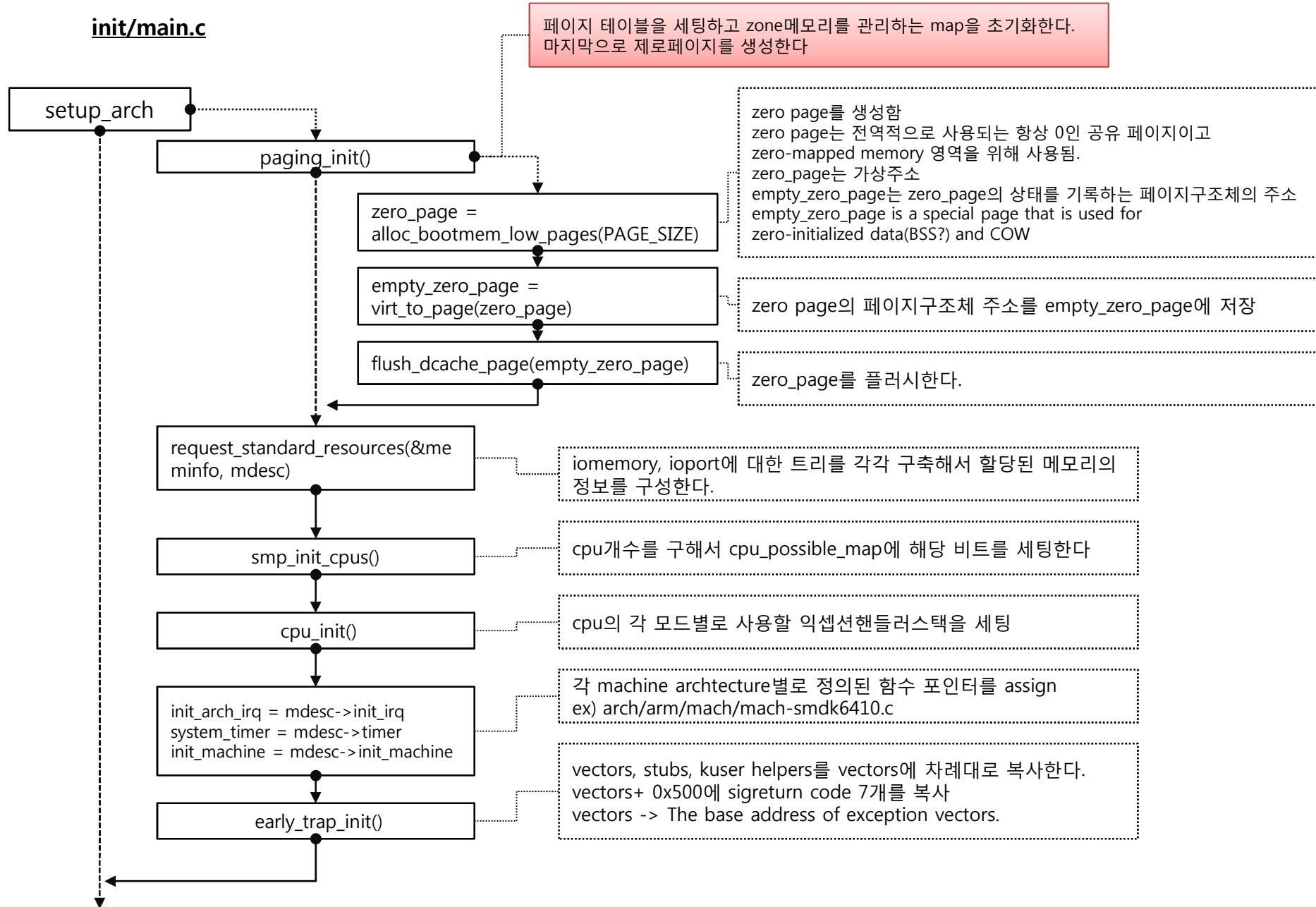
update된 부분을 반영하기 위해서 tlb & cache를 flush함
flush_cache_all() 함수가 호출될때, 실제 구현된 함수는
arch/arm/mm/cache-*.S and arch/arm/mm/proc-*.S files에 존재함.

KConfig에서 CONFIG_HIGHMEM을 셋팅함
CONFIG_HIGHMEM이 정의되어 있지않으면 아무일도 수행하지 않음.

ARM에서 High Memory Support는 Experimental이고 여기에서
정의하는 HIGH MEM Support는 메모리가 4GB보다 큰 메모리를
시스템이 가지고있을 때 지원하기 위한 것임

0xffff0000은 high-vectors가 위치한 가상주소. 이 주소가
top_pmd임

init/main.c

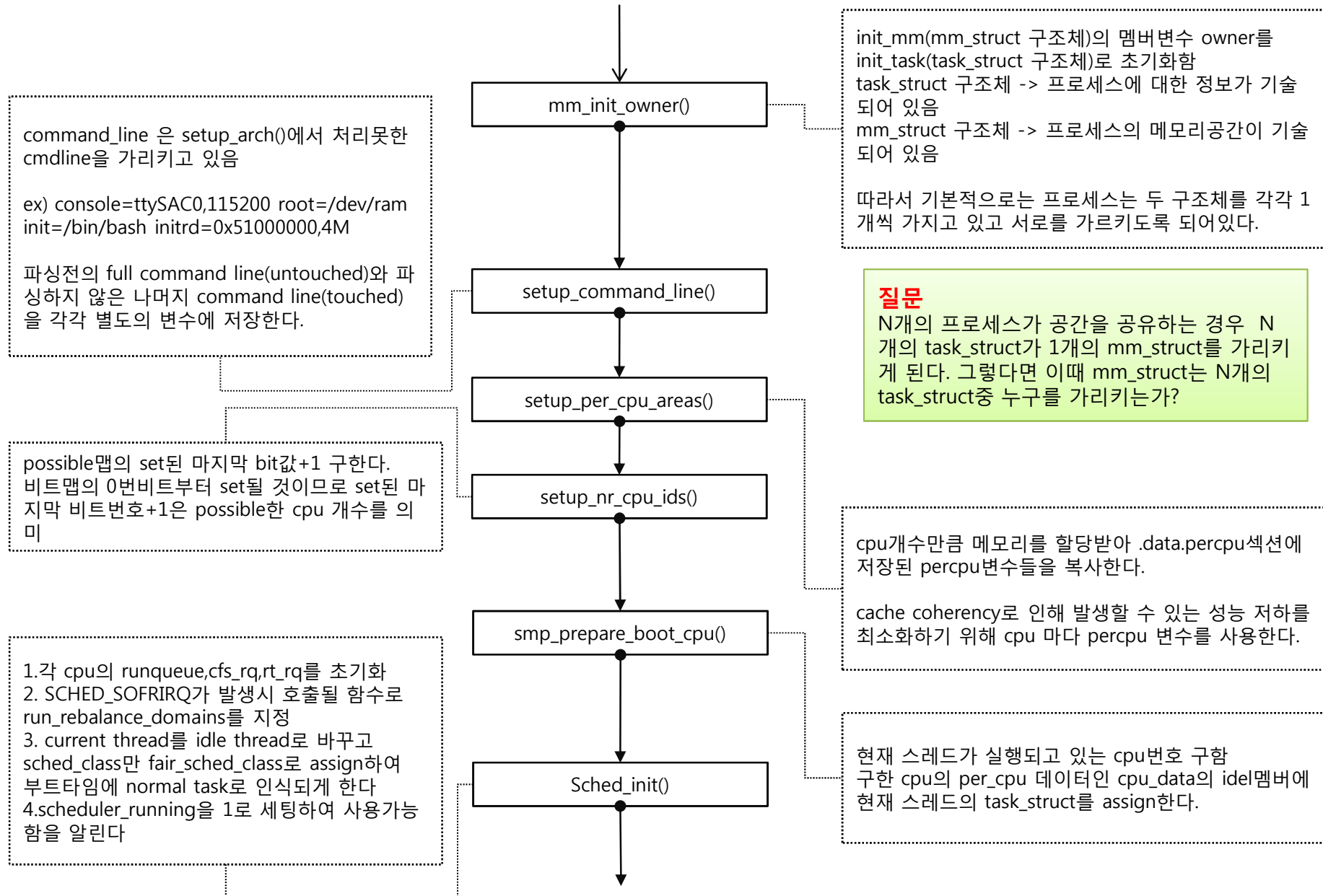


init/main.c

setup arch 함수의 주요 Operation

1. 부트로더에서 넘어온 atag를 각 항목별로 파싱함수를 콜해서 처리 atag는 램디스크정보, cmdline, 메모리정보, 페이지크기등을 가지고 있다.
2. machine architecture nr을 이용해서 해당되는 machine descriptor 구조체를 구한다.
3. 부팅을 진행하는 프로세스의 init_mm(mm_struct형 구조체)에 가상메모리구조에 대한 정보를 설정
4. 페이지 테이블을 세팅, 시스템의 메모리를 관리하는 맵을 초기화함
5. iomem, ioport에 대한 트리를 각각 구축해서 할당된 메모리의 정보를 계층구조로 구성
6. cpu의 각 모드별로 사용할 익셉션핸들러 스택을 세팅
7. vector테이블에 핸들러코드등을 복사

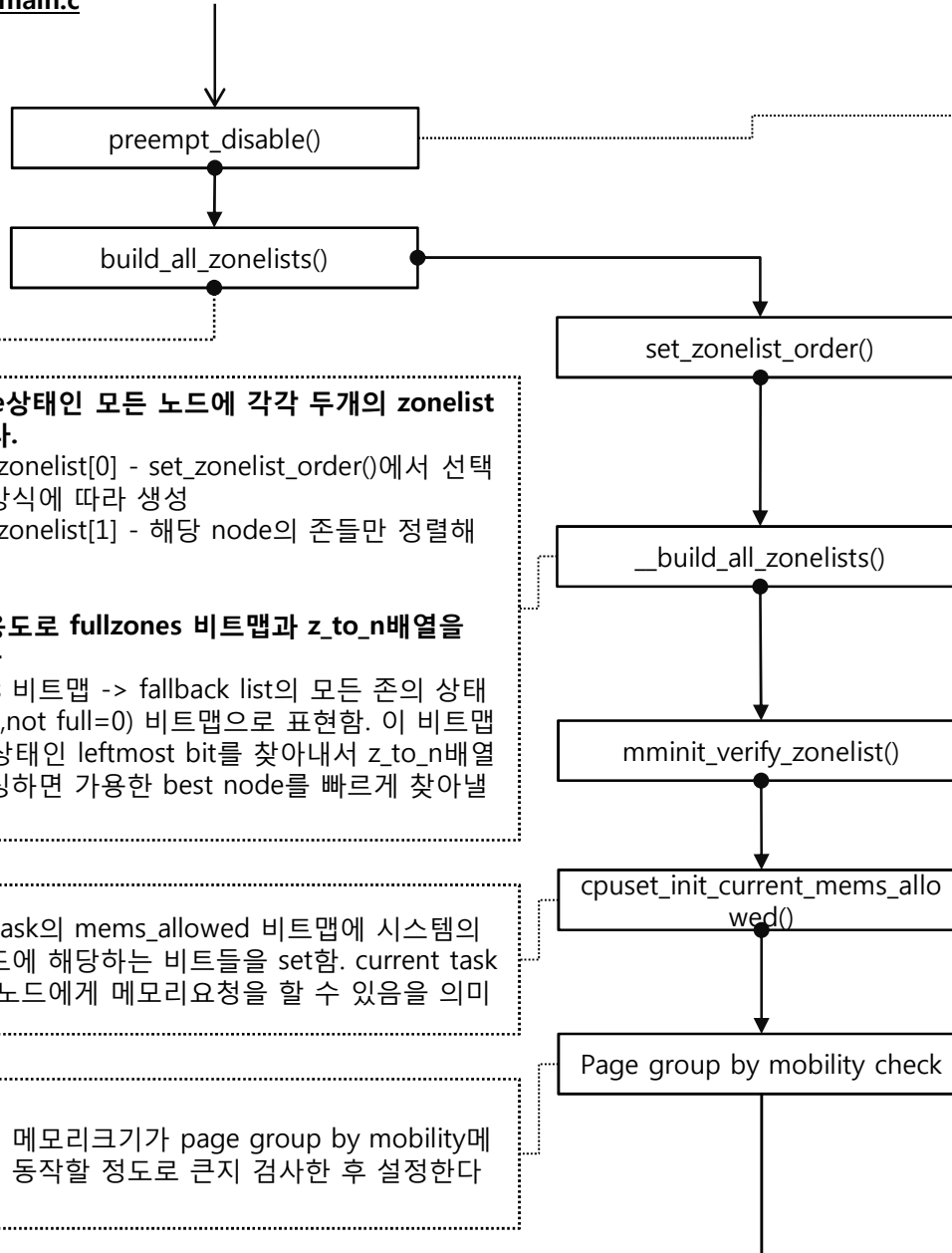
init/main.c



Scheduling Policy

Todo: 스케줄링 방법,
대상에 따른 정책 정리

init/main.c



선점을 비활성화 한다.

build_all_zonelist()에서 생성할 zonelist의 정렬방식을 결정한다.

ZONE ORDER -
NODE0.ZONE_NORMAL,
NODE1.ZONE_NORMAL,
NODE2>ZONE_NORMAL,
NODE0.ZONE_DMA,
NODE1.ZONE_DMA...

장점 = OOM문제발생이 쉽게 되지 않음,
단점 = best locality를 제공하지 못함

NODE ORDER -
NODE0.ZONE_NORMAL,
NODE0.ZONE_DMA,
NODE1.ZONE_NORMAL,
NODE1.ZONE_DMA,
NODE2.ZONE_NORMAL...

장점 = best locality
단점 = OOM문제가 발생할 소지가 있음

따라서 DMA,DMA32존의 전체페이지가 시스템 전체페이지의 절반 이상이라면 OOM문제에서 상대적으로 안전하다고 보고 NODE ORDER방식을 선택하고, 절반 이하일 때는 ZONE ORDER 방식을 선택한다.

시스템의 모든 존의 정보를 출력한다.
이때 online인 노드에 속하고 present page가 존재하는 존을 대상으로 한다.

그리고 command line으로 mminit_loglevel 값이 MMINIT_VERIFY이상으로 넘어왔을때만 출력한다

1. online상태인 모든 노드에 각각 두개의 zonelist를 만든다.

pgdat->zonelist[0] - set_zonelist_order()에서 선택된 정렬방식에 따라 생성
pgdat->zonelist[1] - 해당 node의 존들만 정렬해서 생성

2. 캐시용도로 fullzones 비트맵과 z_to_n배열을 생성한다

fullzones 비트맵 -> fallback list의 모든 존의 상태를(full=1,not full=0) 비트맵으로 표현함. 이 비트맵의 clear상태인 leftmost bit를 찾아내서 z_to_n배열을 인덱싱하면 가용한 best node를 빠르게 찾아낼 수 있다.

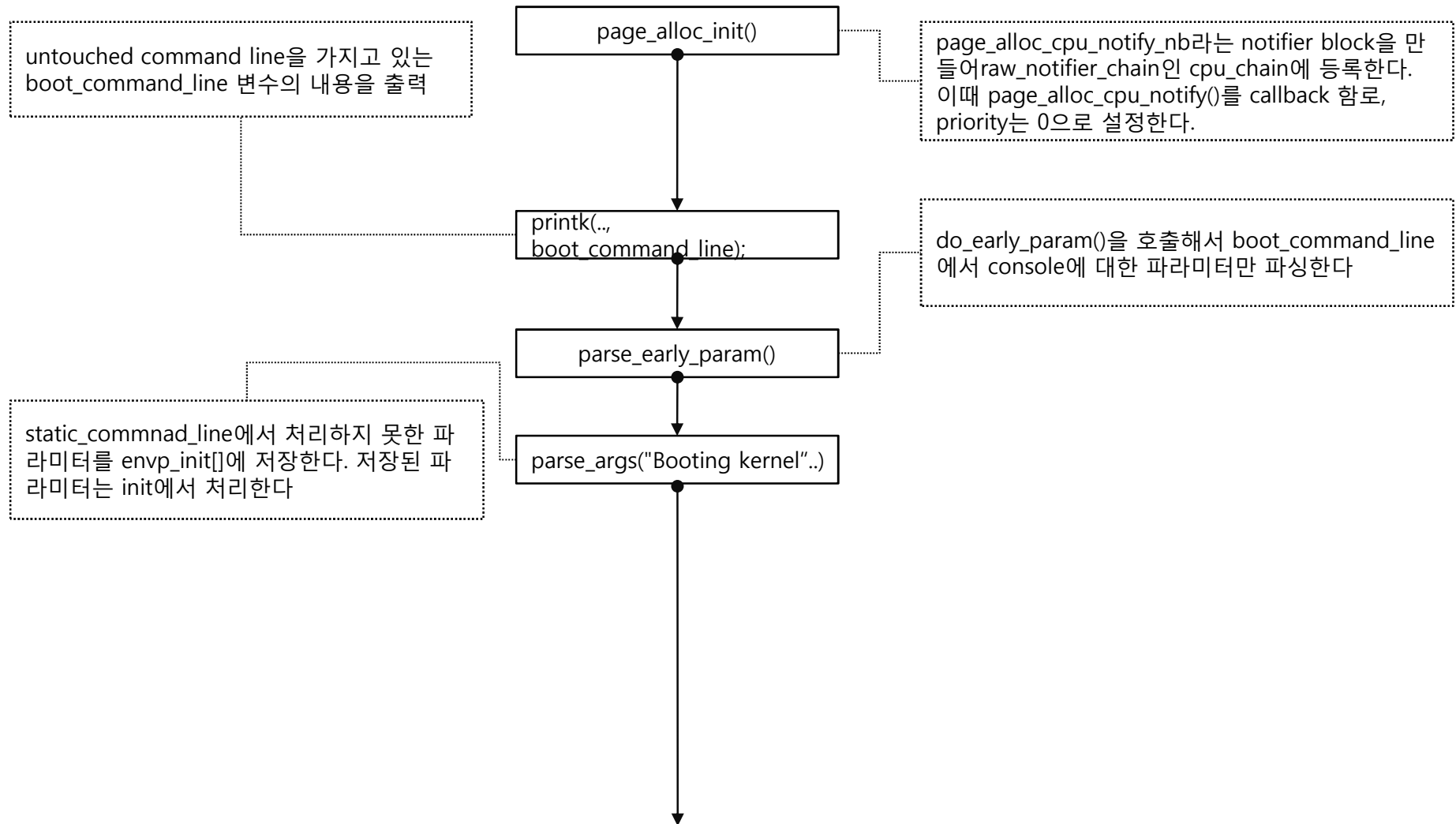
current task의 mems_allowed 비트맵에 시스템의 모든 노드에 해당하는 비트들을 set함. current task가 모든 노드에게 메모리요청을 할 수 있음을 의미

시스템의 메모리크기가 page group by mobility메카니즘이 동작할 정도로 큰지 검사한 후 설정한다

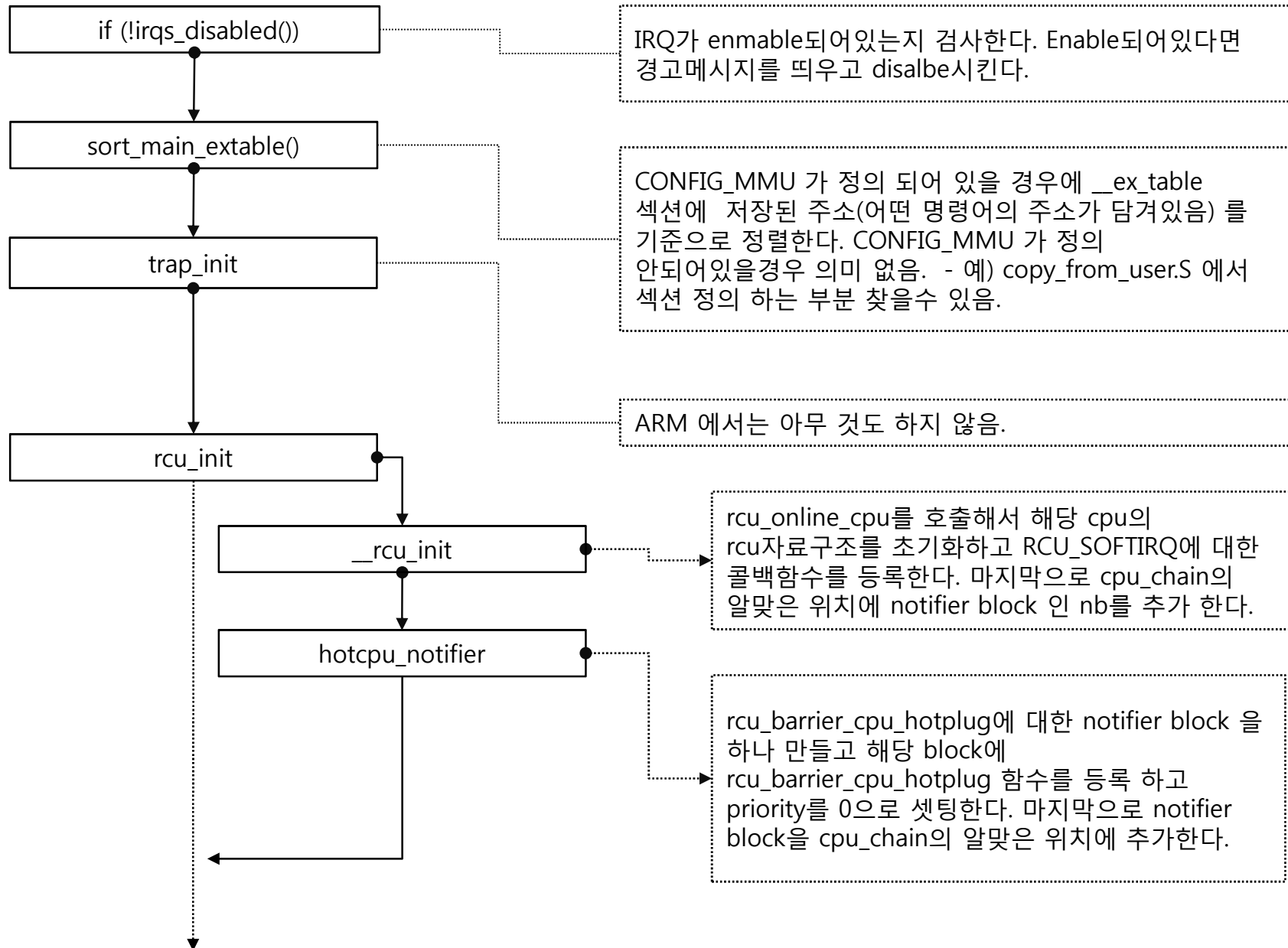
Page group by mobility

Todo: page group by mobility에 대해..

init/main.c



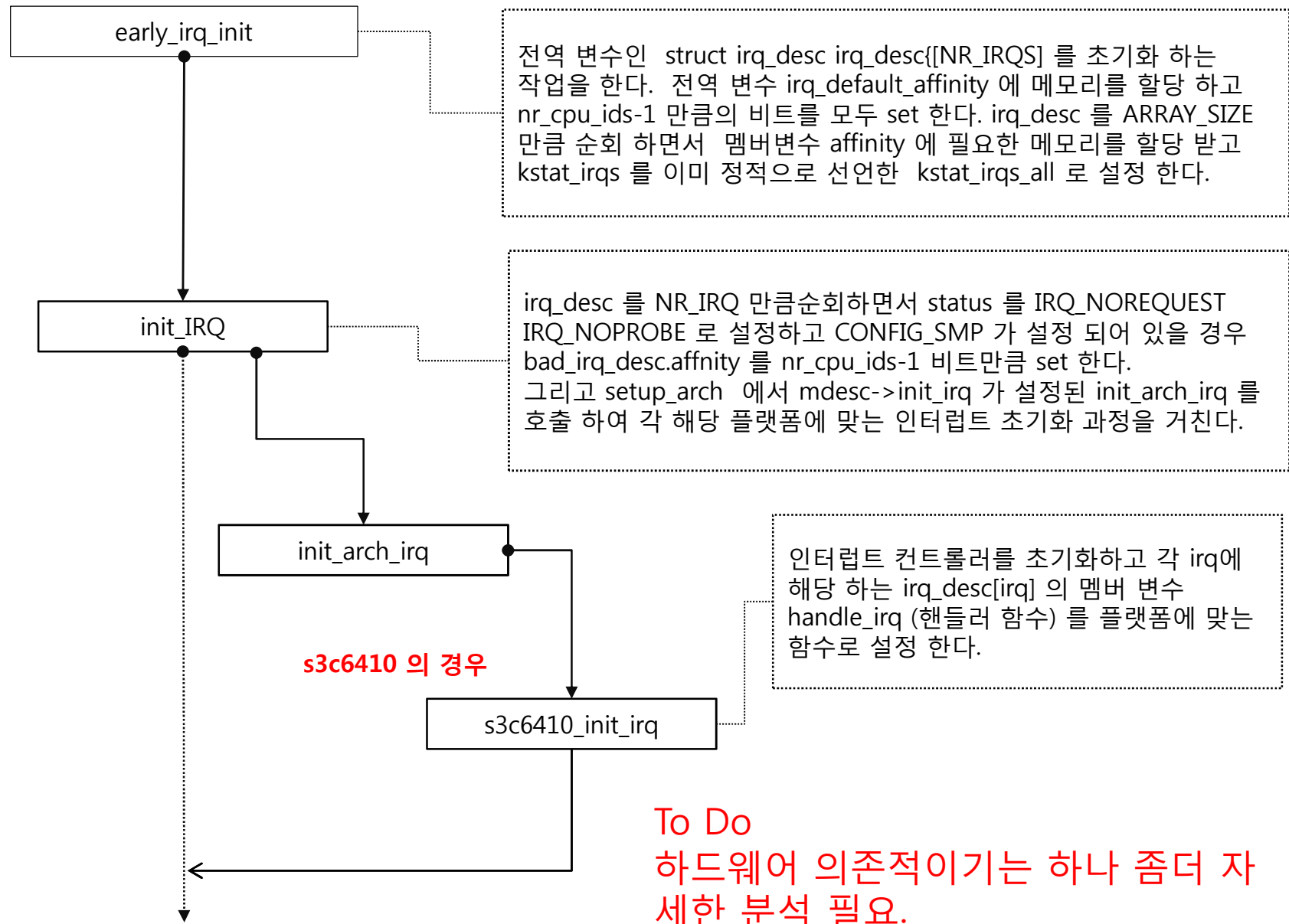
init/main.c



Memory Types

Todo: RCU 개념 정리

init/main.c



irq_desc 구조체와 irq_desc[NR_IRQS] 정적 구조체 배열

kernel/handle.c

```
struct irq_desc irq_desc[NR_IRQS]
__cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .status = IRQ_DISABLED,
        .chip = &no_irq_chip,
        .handle_irq = handle_bad_irq,
        .depth = 1,
        .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
```

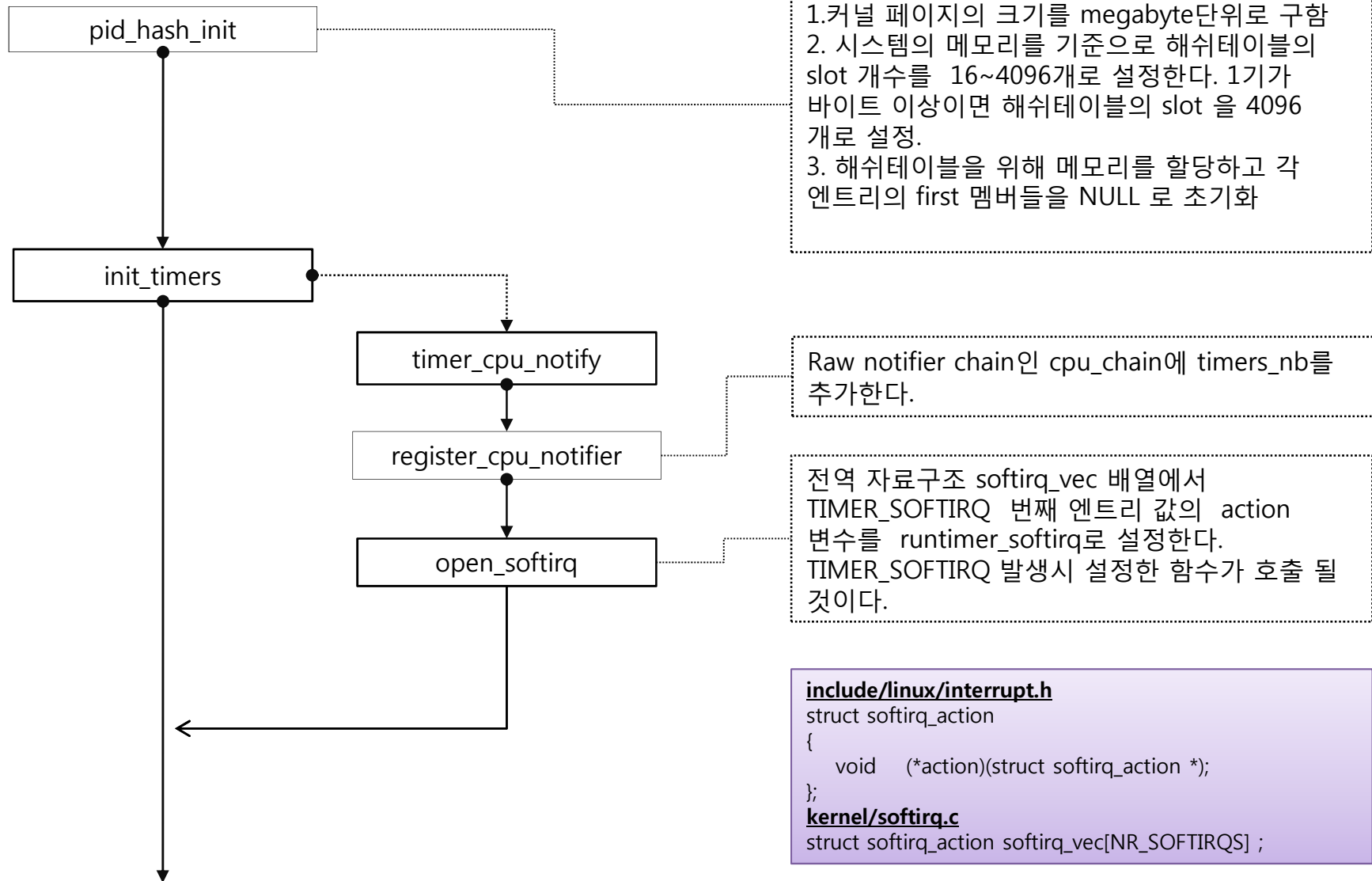
early_irq_init 함수와 init_IRQ 함수에서 참조되는 구조체 이다. 인터럽트 처리를 위해 중요한 자료 구조 이다.

include/linux/irq.h

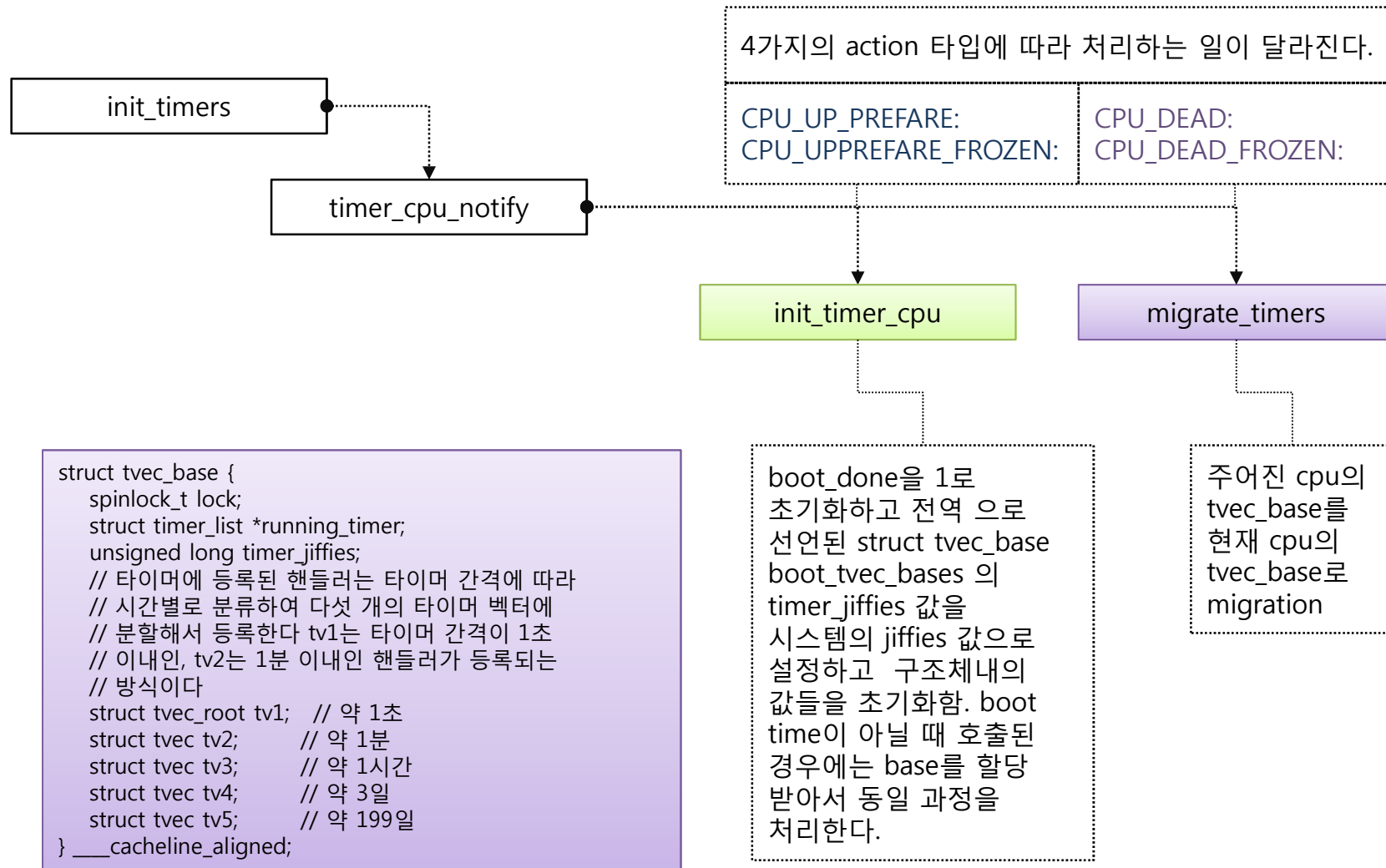
```
struct irq_desc {
    unsigned int      irq;
    struct timer_rand_state *timer_rand_state;
    unsigned int      *kstat_irqs;
#ifdef CONFIG_INTR_REMAP
    struct irq_2_iommu   *irq_2_iommu;
#endif
    irq_flow_handler_t handle_irq;
    struct irq_chip      *chip;
    struct msi_desc      *msi_desc;
    void                *handler_data;
    void                *chip_data;
    struct irqaction     *action; /* IRQ action list */
    unsigned int         status; /* IRQ status */

    unsigned int         depth; /* nested irq disables */
    unsigned int         wake_depth; /* nested wake enables */
    unsigned int         irq_count; /* For detecting broken IRQs */
    unsigned long         last_unhandled; /* Aging timer for unhandled count */
    unsigned int         irq_unhandled;
    spinlock_t           lock;
#ifdef CONFIG_SMP
    cpumask_var_t        affinity;
    unsigned int         cpu;
#endif
#ifdef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t        pending_mask;
#endif
#ifdef CONFIG_THREADS_ACTIVE
    atomic_t             threads_active;
    wait_queue_head_t     wait_for_threads;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry *dir;
#endif
    const char           *name;
} ____cacheline_internodealigned_in_smp;
```

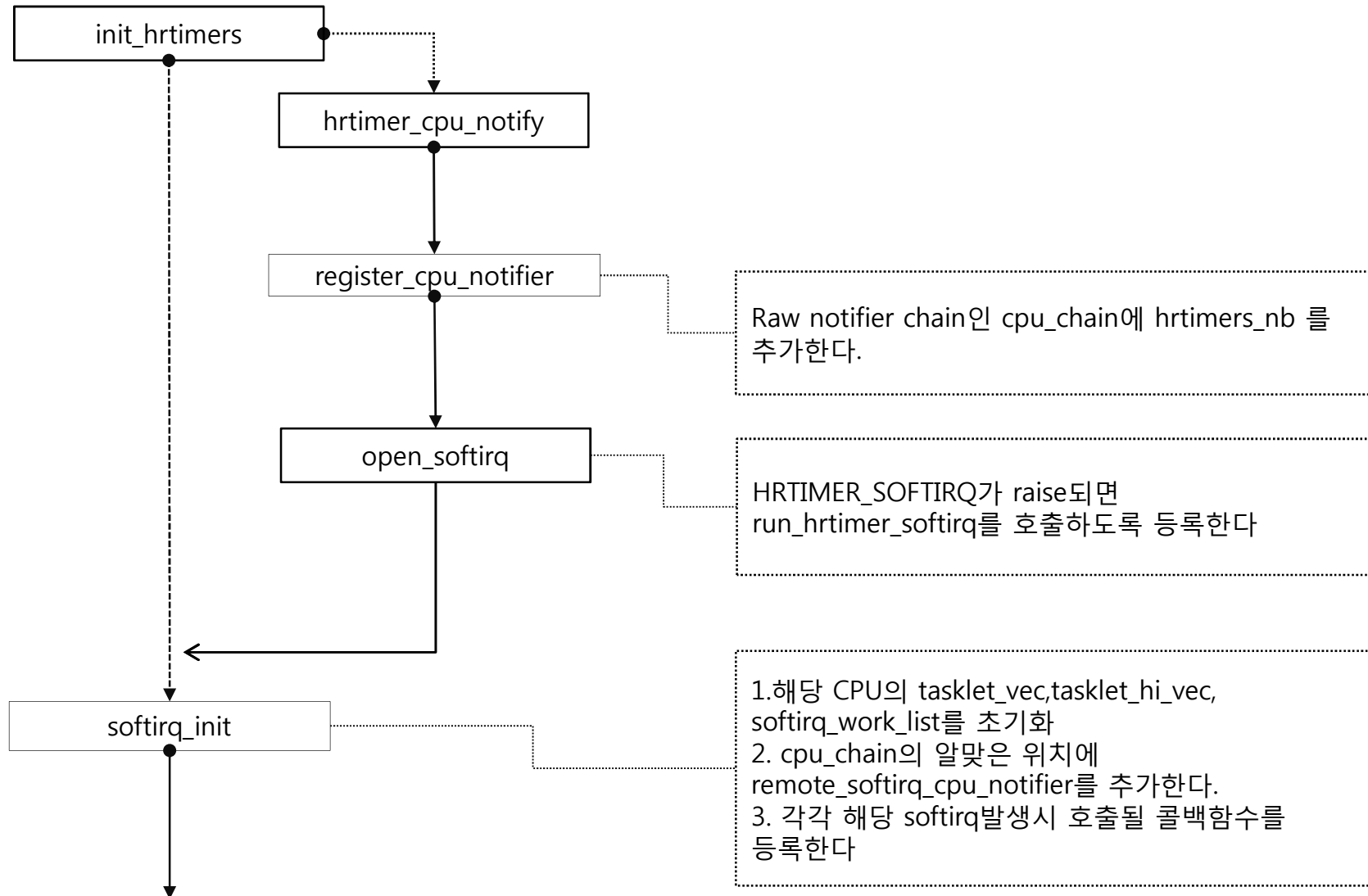
init/main.c



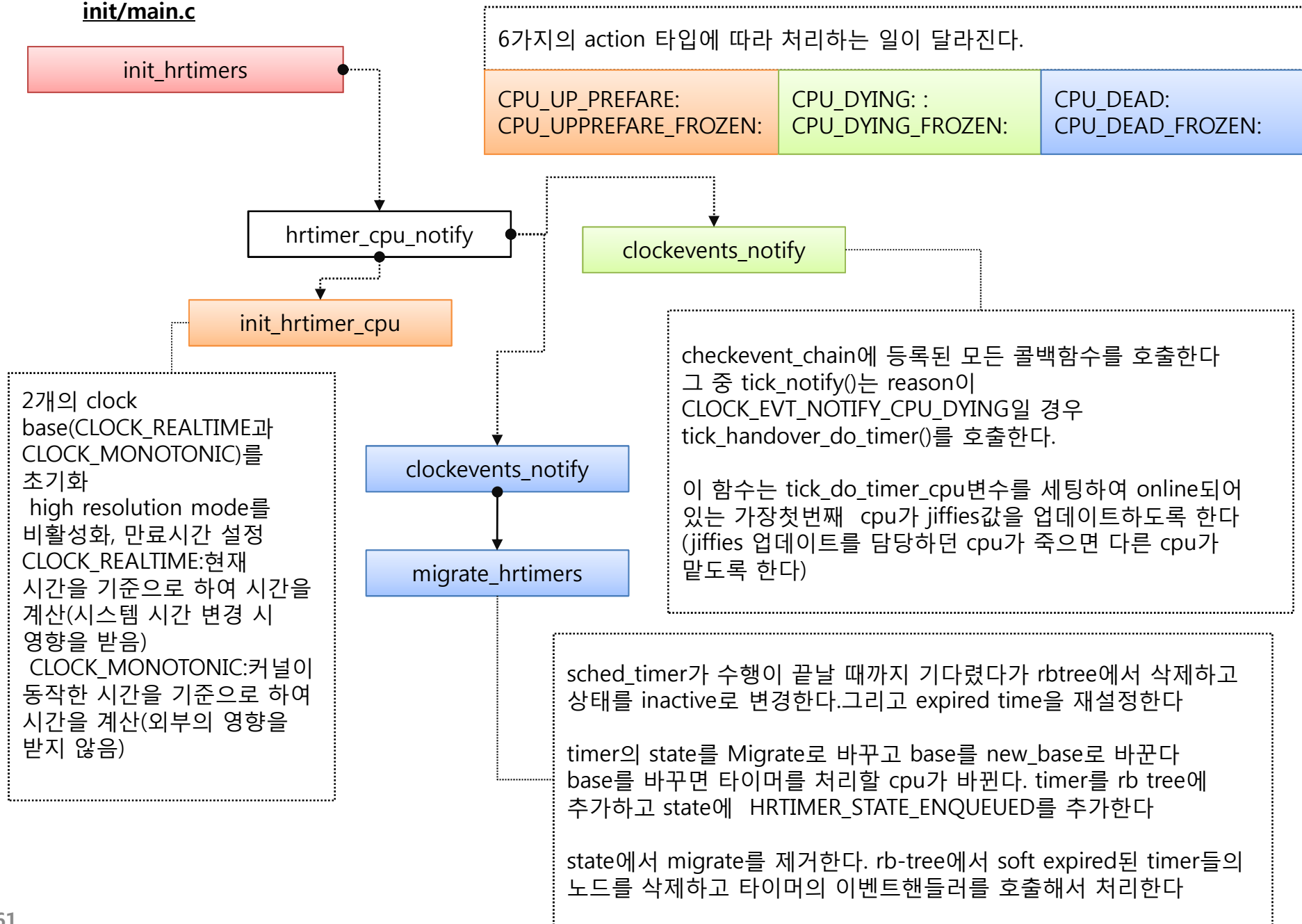
init/main.c



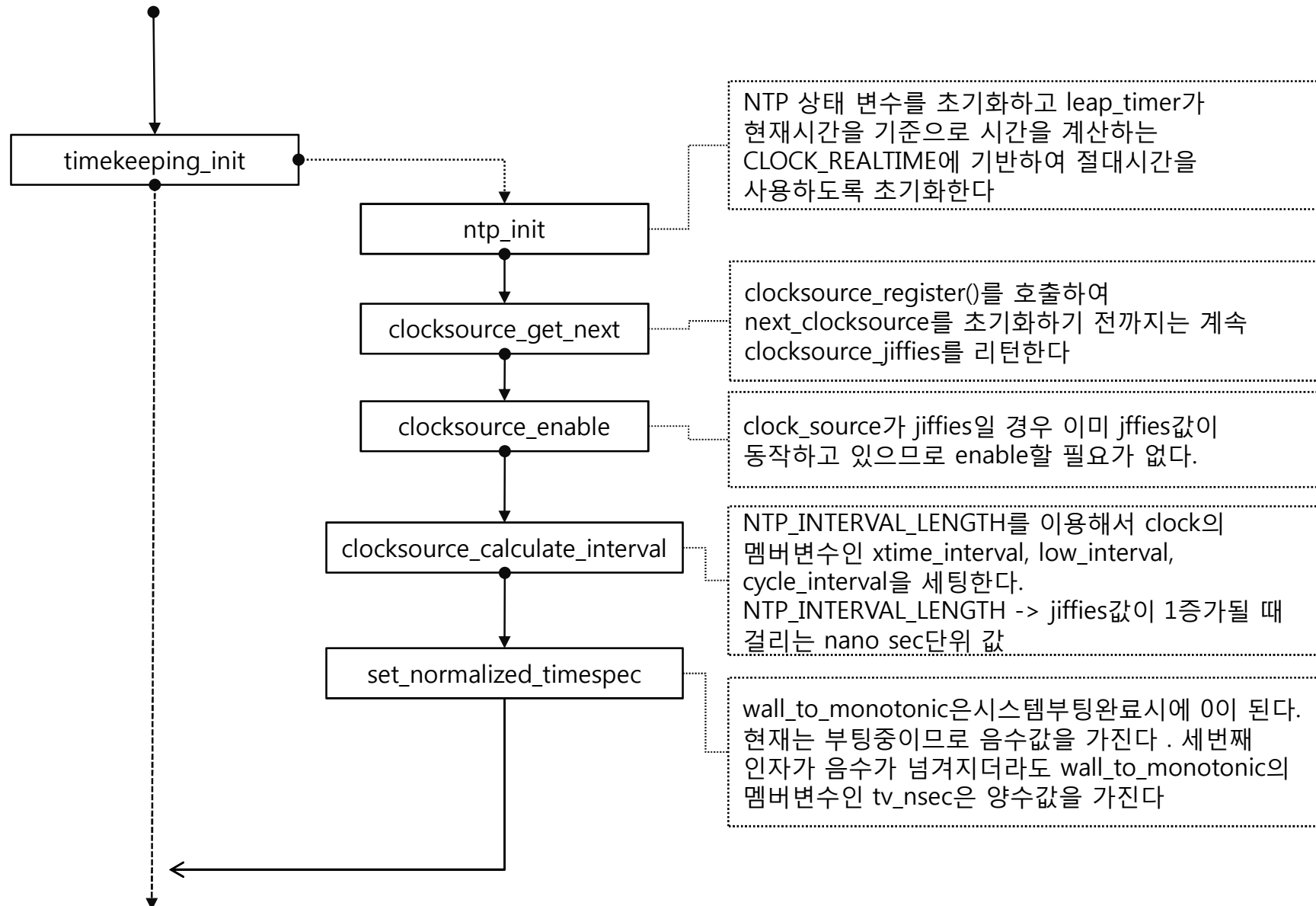
init/main.c



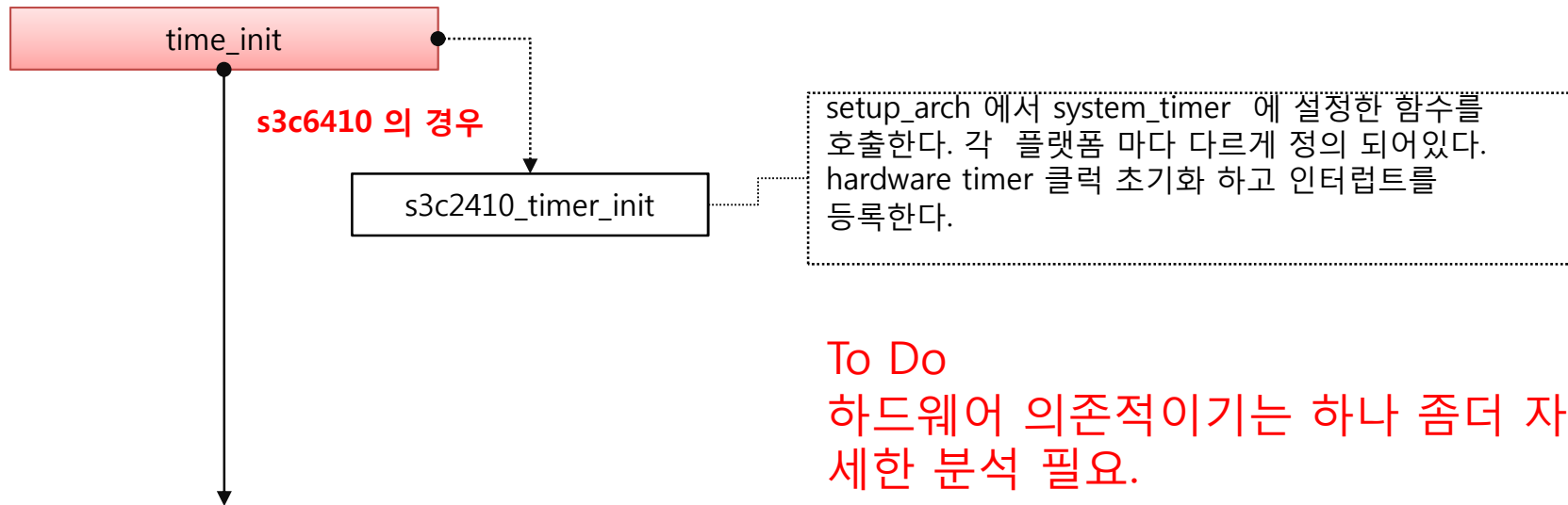
init/main.c

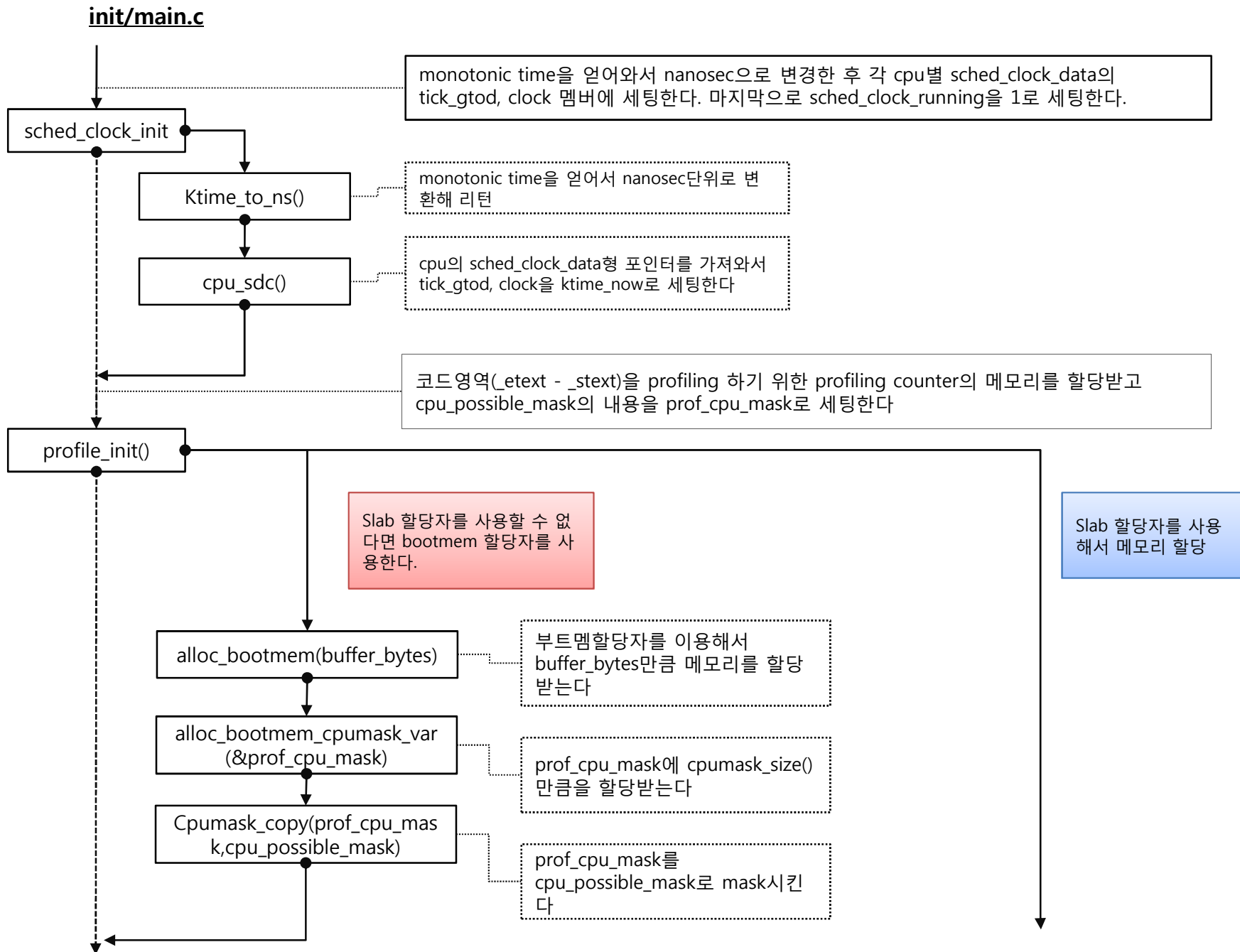


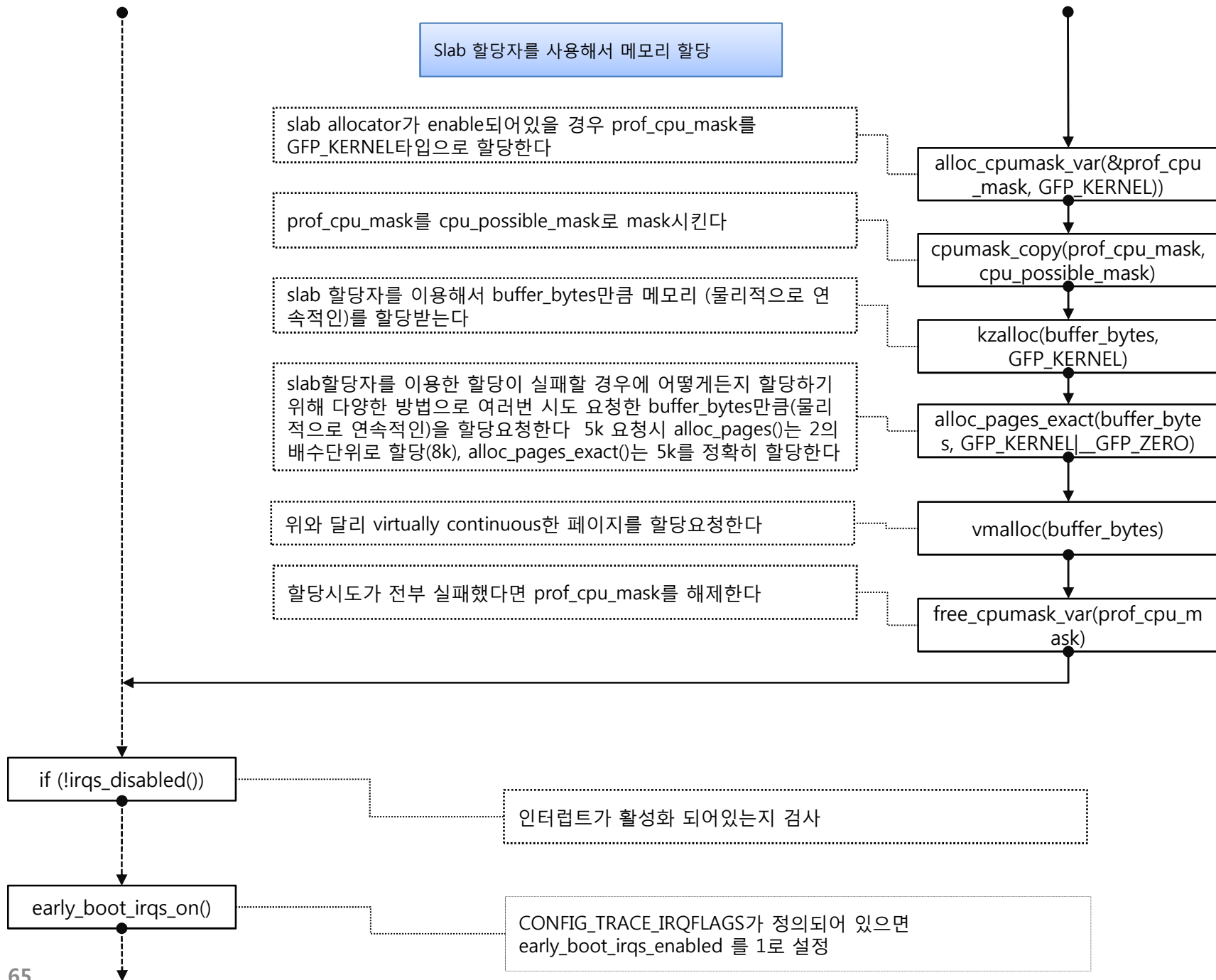
init/main.c

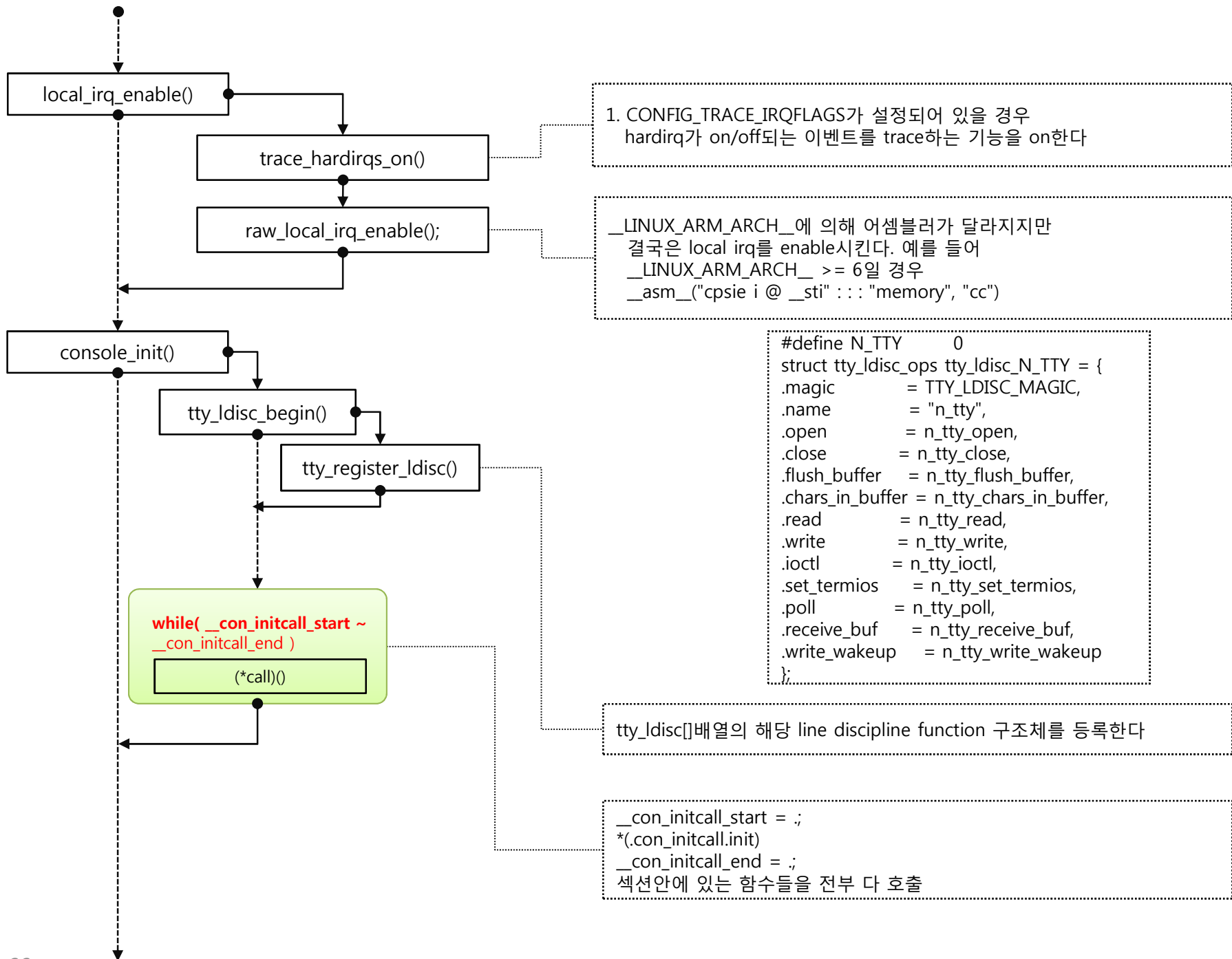


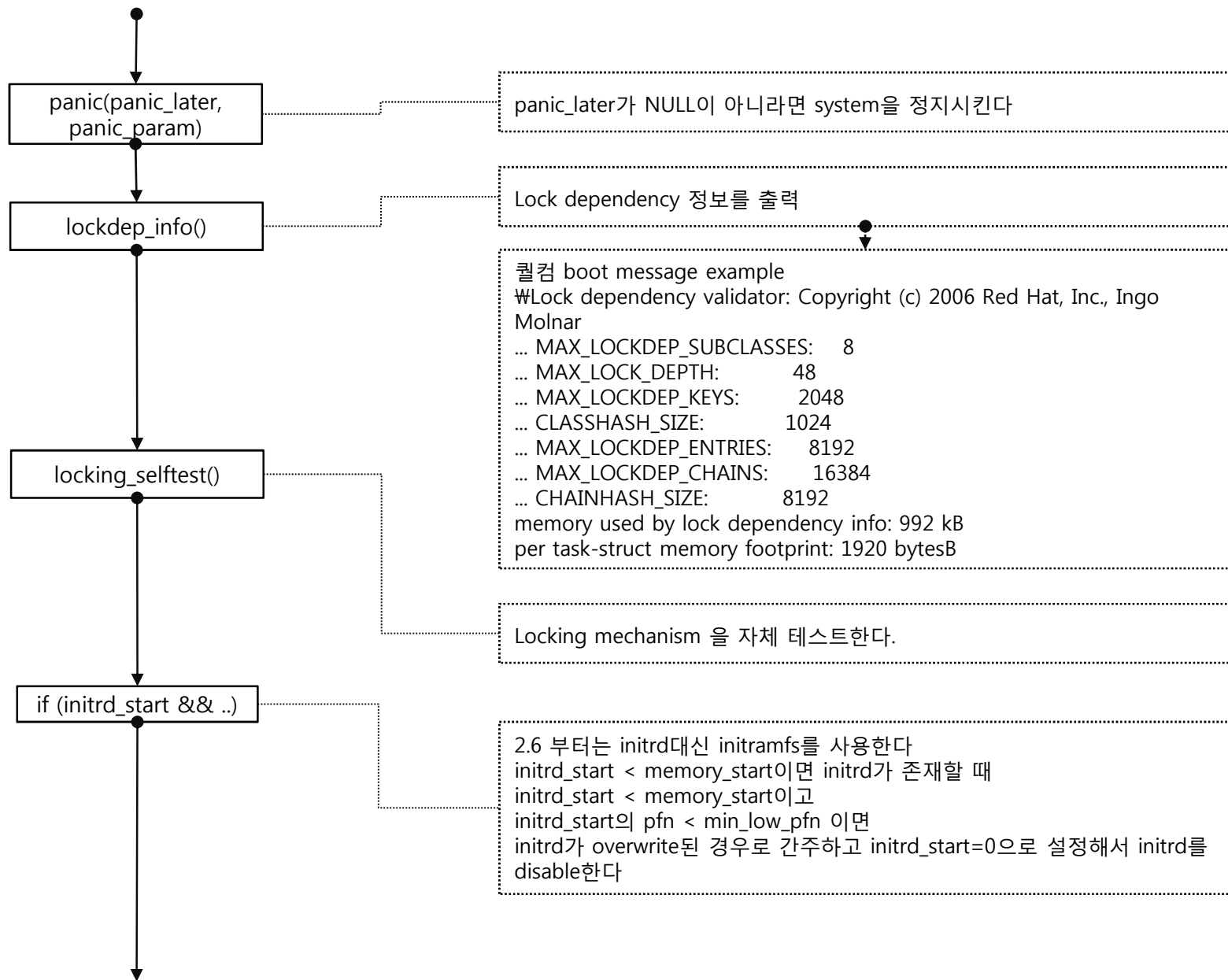
init/main.c



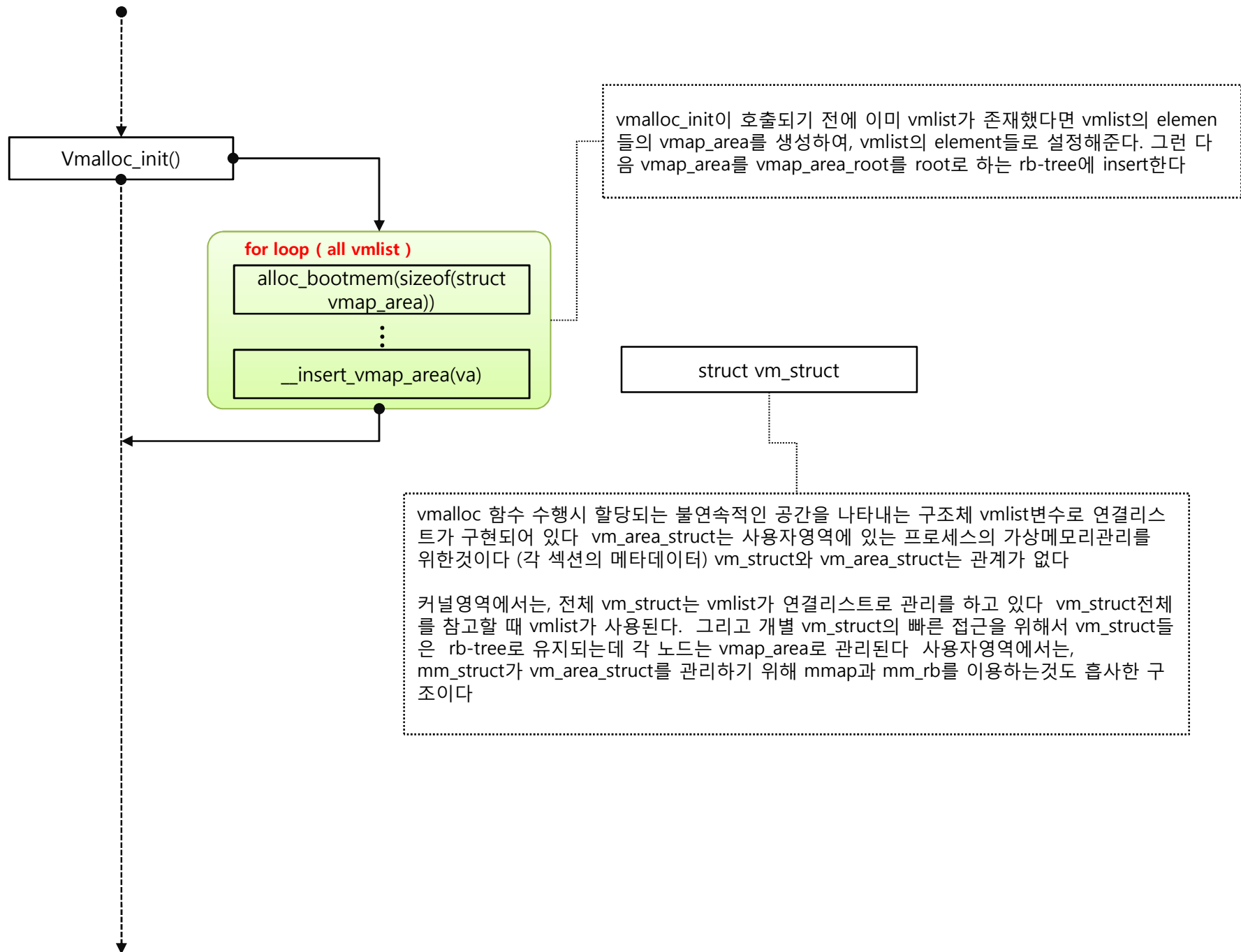


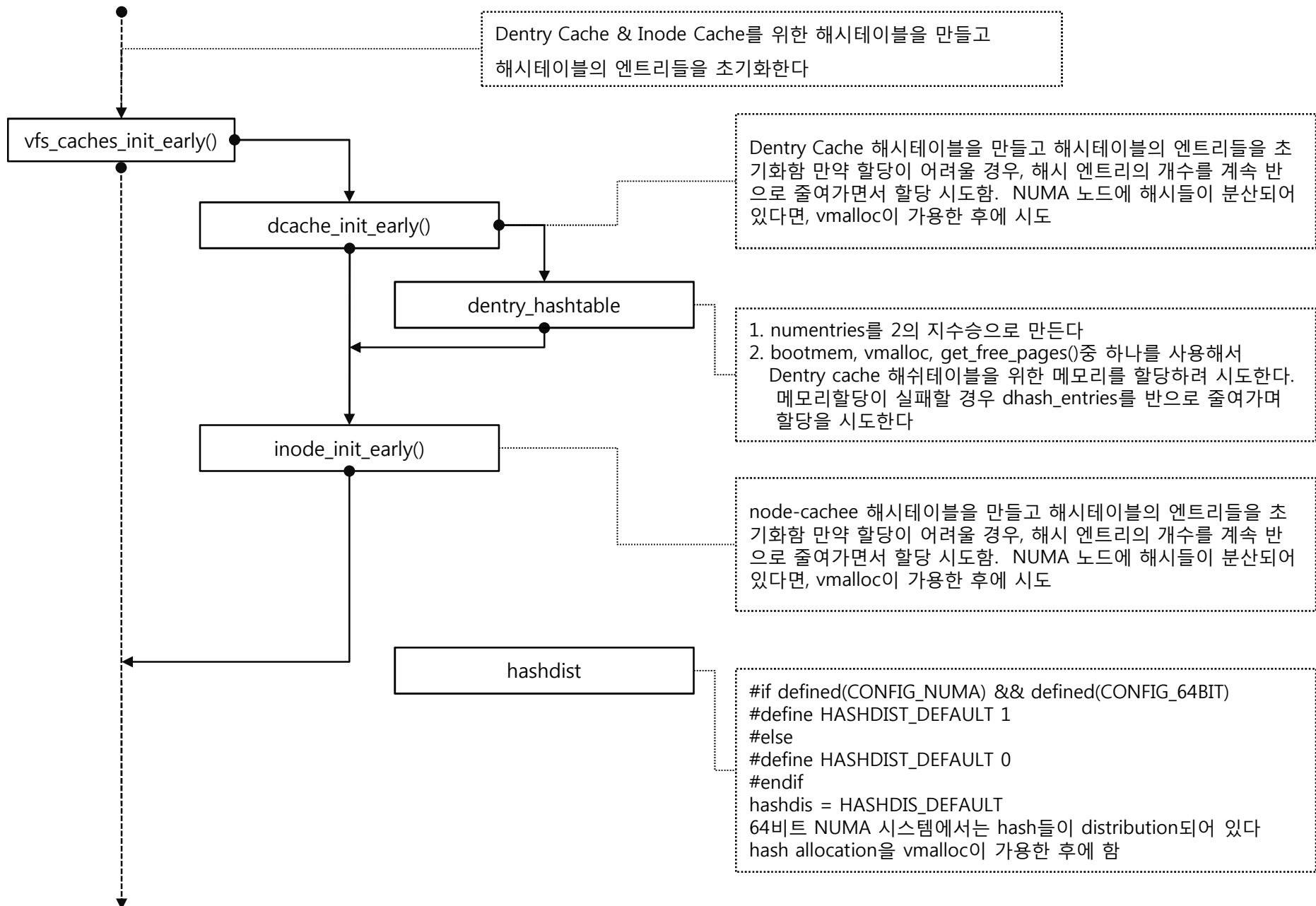




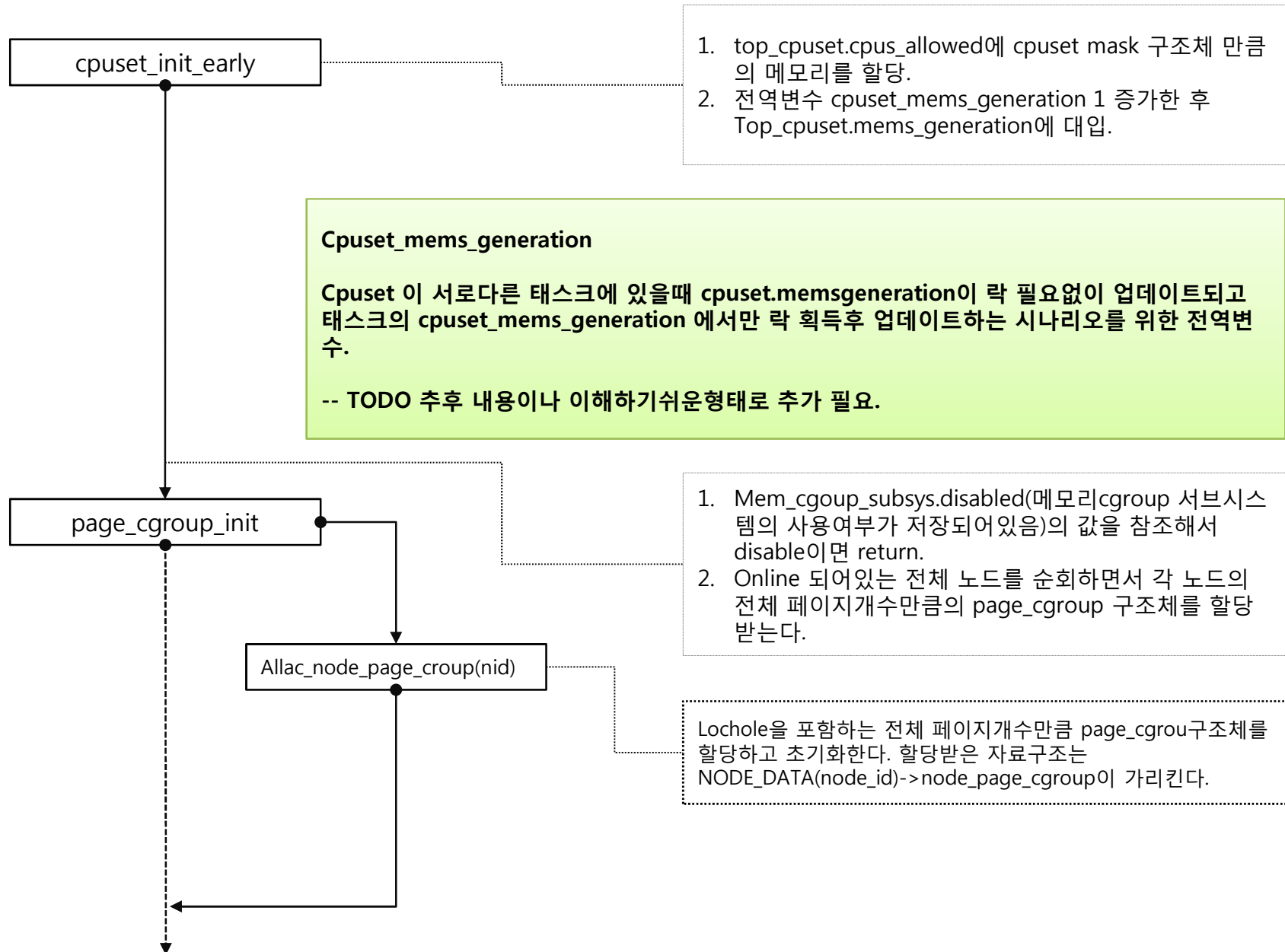


TODO : lockdep 개념정리

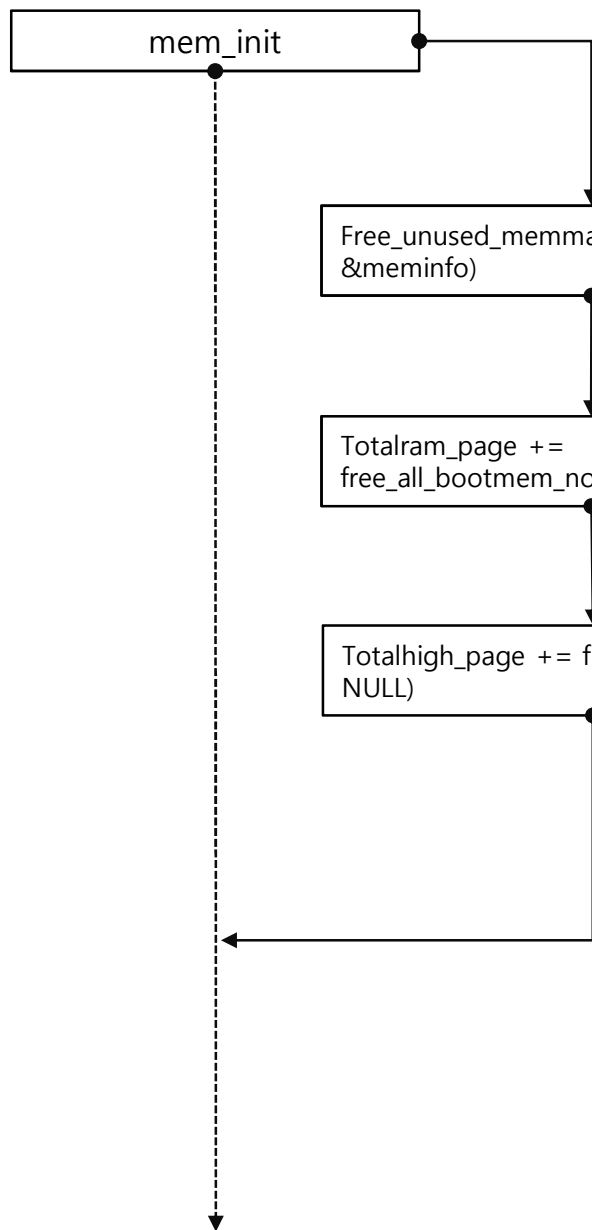




init/main.c



init/main.c



1. Mem_map에서 free area를 마킹하고 free memory가 얼마인지 알려줌.
2. 각 노드의 뱅크를 순회하면서 뱅크사이의 홀에 대한 영역을 free해줌.
3. Bitmap의 bit가 0으로 셋팅된 page를 Buddy Allocator로 릴리즈 함.
4. 시스템의 가용 메모리와 코드, 데이터사이즈등 정보표시.

두 뱅크사이에는 hole이 존재할 수 있는데 이 영역도 페이지 구조체가 할당되어 있다. 해당 page 구조체를 free함

Node의 free page를 buddy allocator에 release시킴.

1. 현재 노드의 free가능한 page를 free시킴.
 - 1) 일반 page인 경우는 free할 위치부터 bulk(order를 기준으로) free할 수 있으면 free하고 bulk로 free할 수 없는 경우는 한 page씩 free시킴.
 - 2) metadata를 가지고 있는 page인 경우, 즉 bitmap이나 frame 정보등. 이 경우는 meta data를 가지고 있는 page를 구하여 해제시킴.
2. Page들이 해제될때 buddy system이 관여되어 해당 정보가 기록됨.
3. 총 free된 page의 수를 리턴.

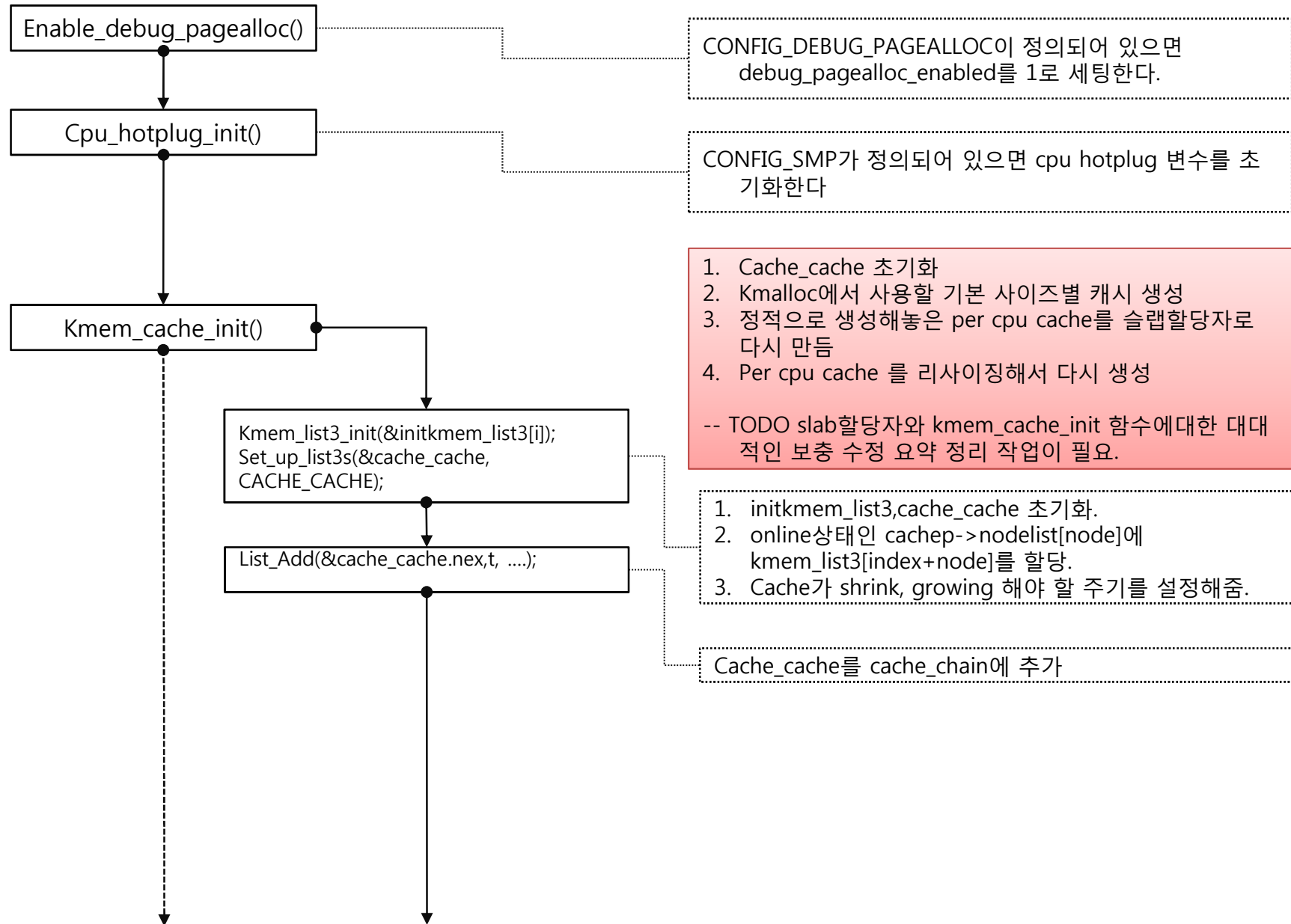
Highmem page를 free시킨다.
참조할부분 :arch/arm/mm/mmu.c::sanity_check_meminfo()
자세한 메커니즘 TODO. 아직 파악안됨.

Sysctl_overcommit_memory

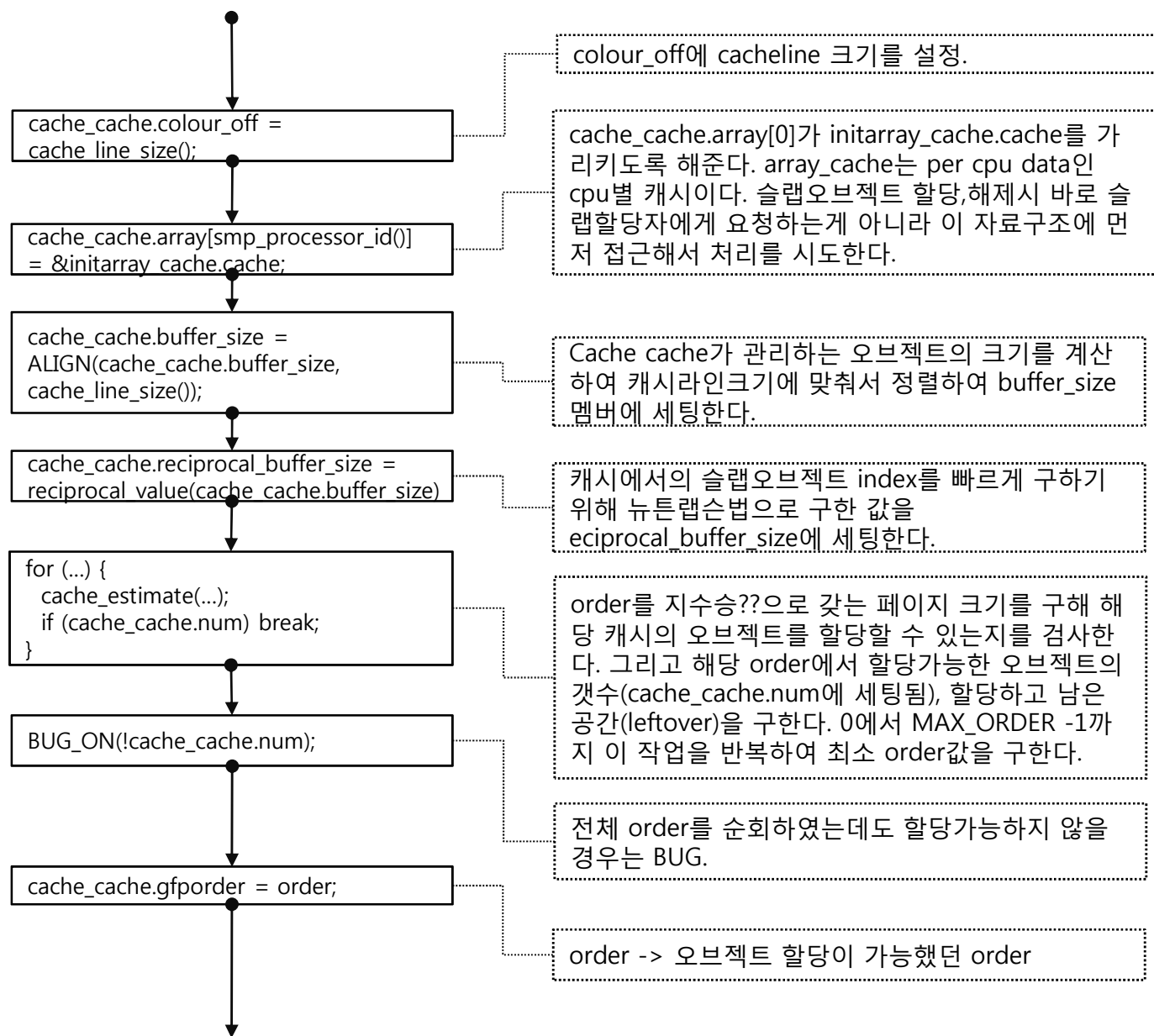
Memory overcommit 정책을 OVERCOMMIT_ALWAYS 로 설정한다.

-- Overcomit에 관하여 자세한 것은 TODO.

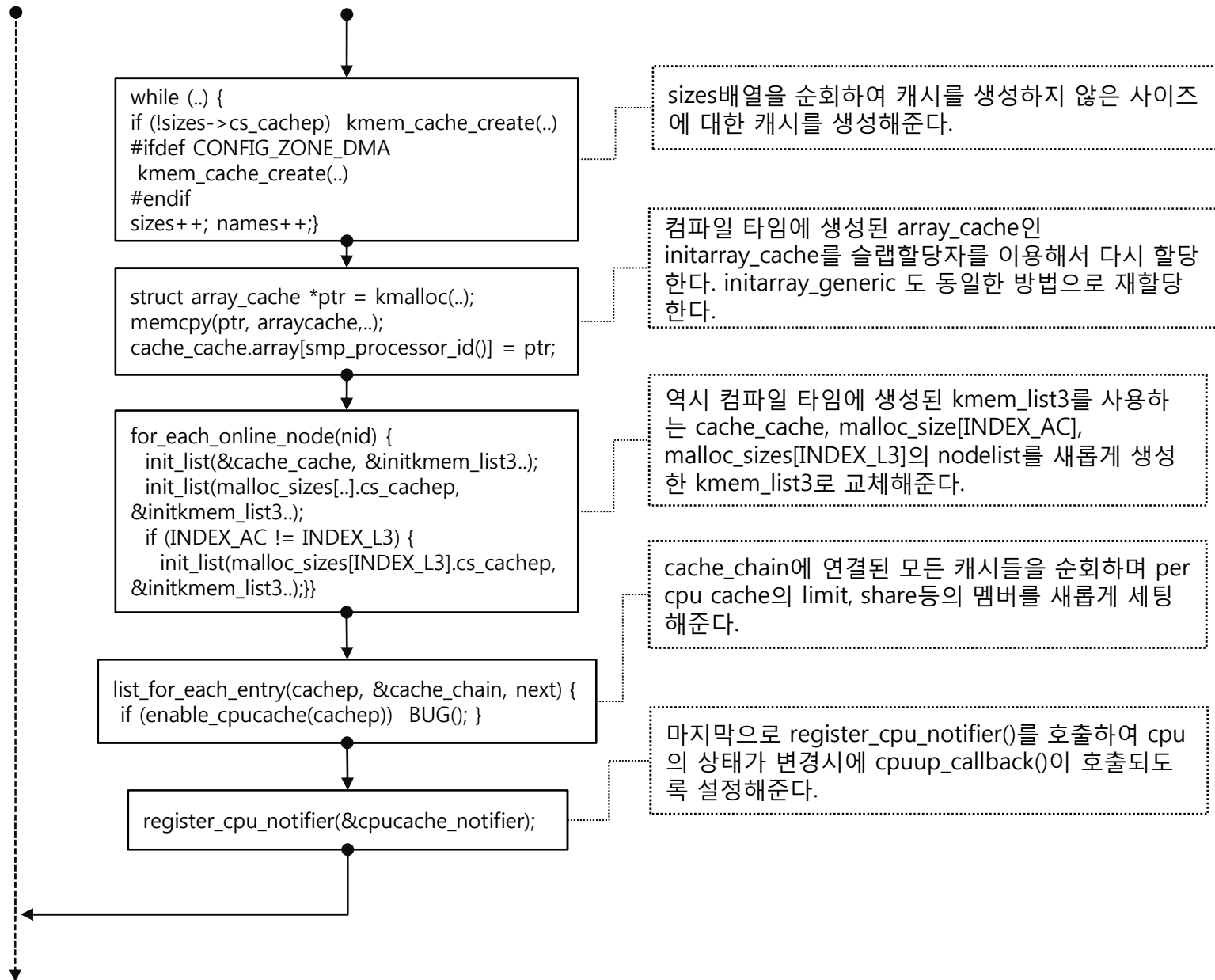
init/main.c



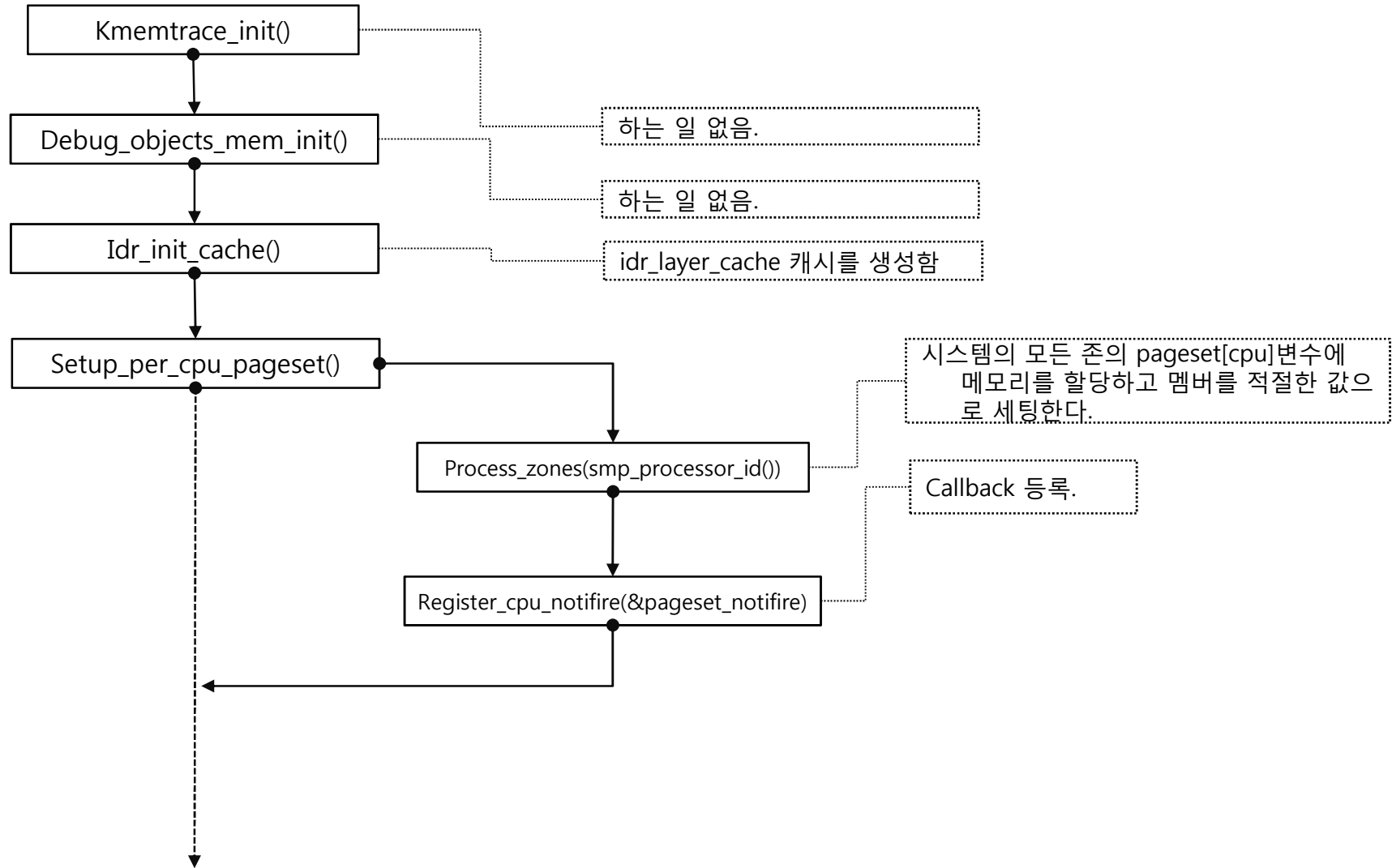
init/main.c



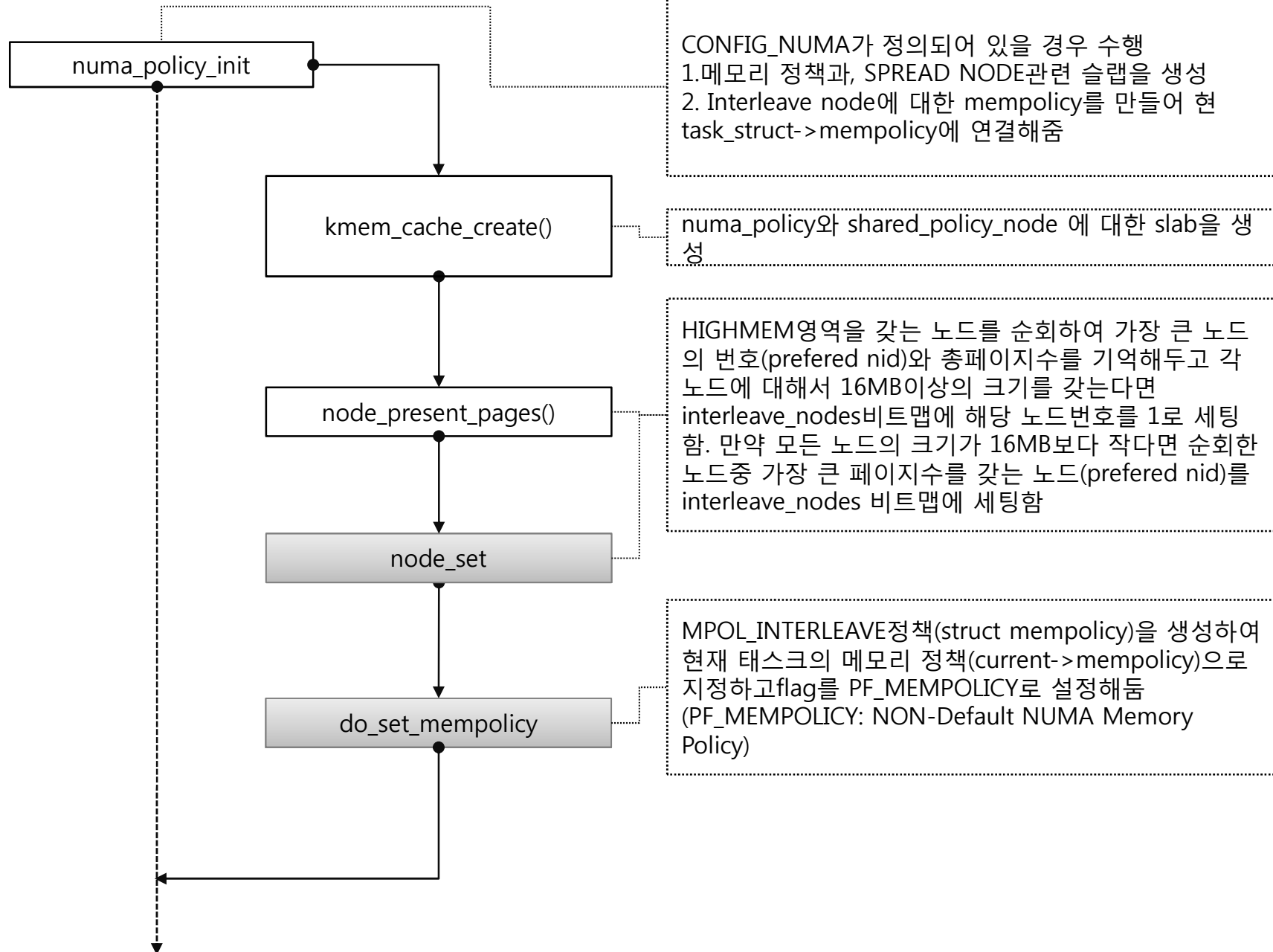
init/main.c



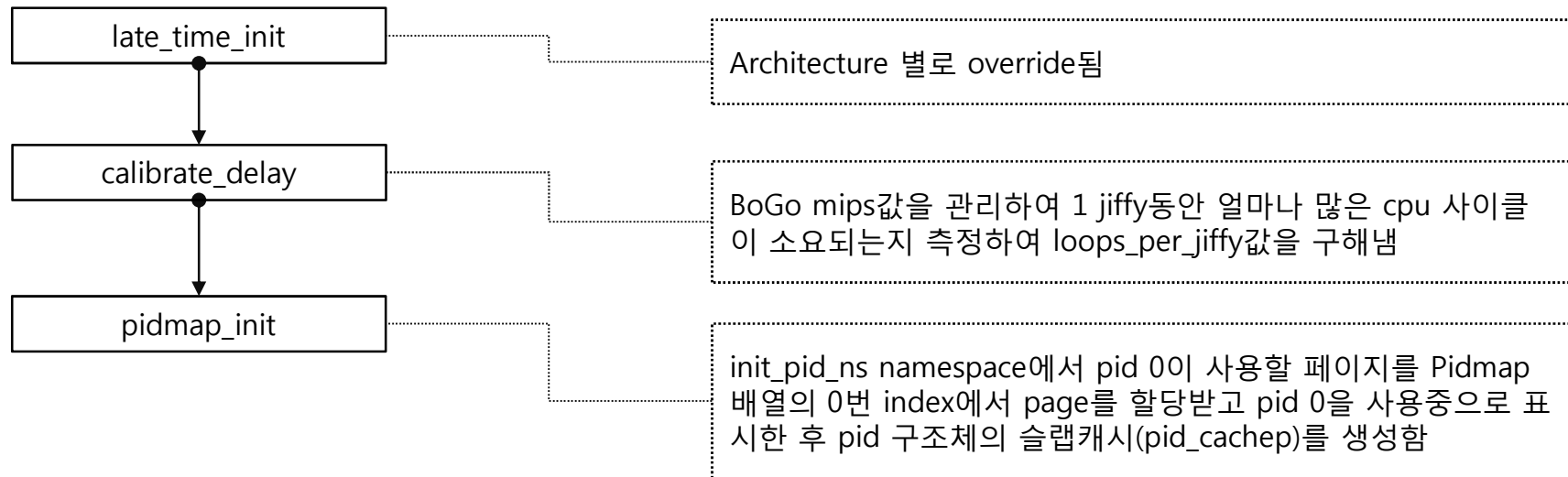
init/main.c



init/main.c



init/main.c



Kernel이 BogomIPS값을 결정하기위한 함수 BogoMIPS

MIPS는 Millions of Instruction Per Second의 약자이다. 계산 능력을 나타내는 지표로 사용된다. BogoMIPS는 리눅스 토발즈가 만들어낸 것으로서, 커널은 부팅시에 현 시스템에서 busy loop이 얼마나 빠른지에 대해 기록한 값이며 Bogo는 bogus(가짜)를 의미한다.

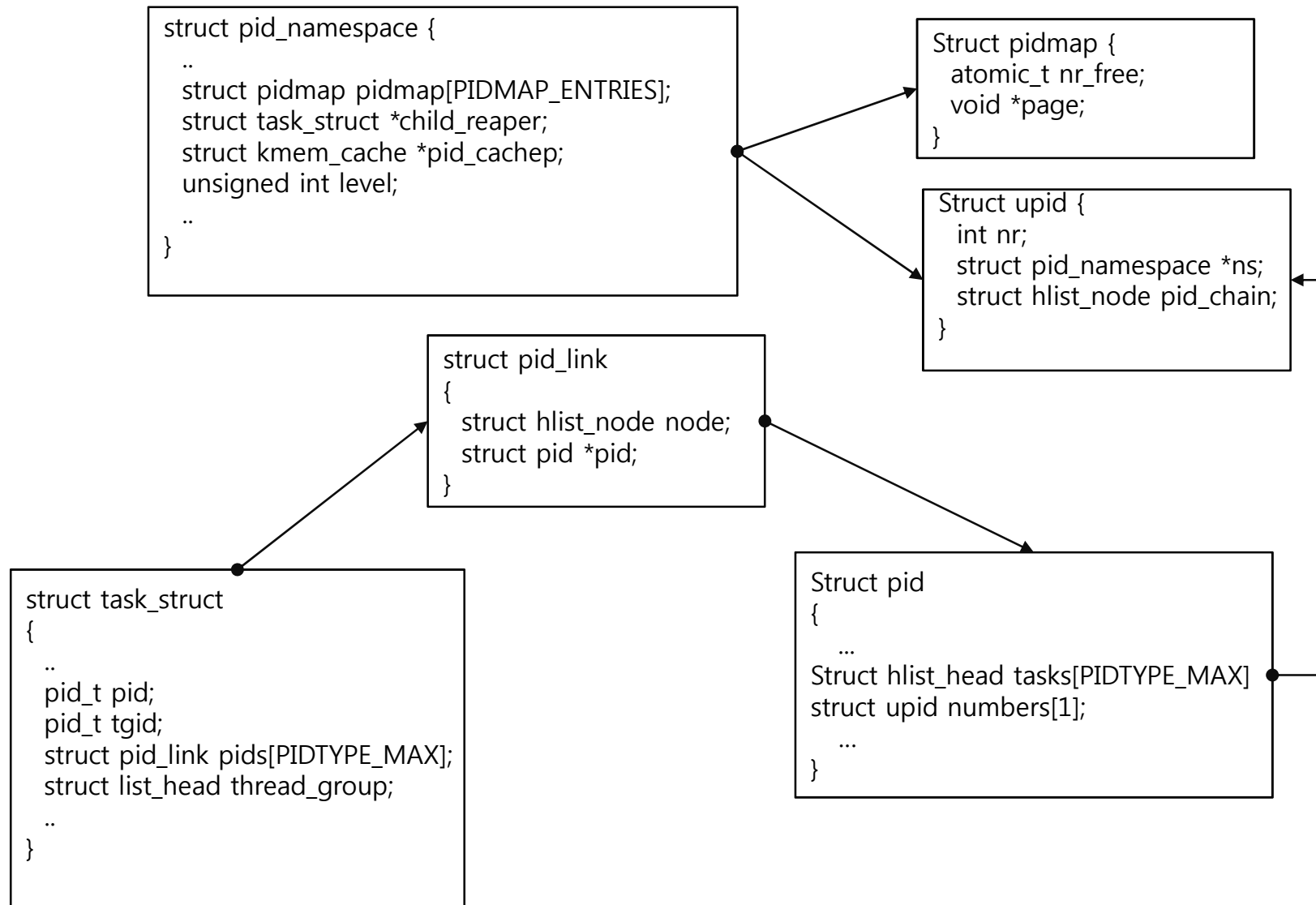
이렇게 측정된 bogoMIPS를 가지고 프로세서의 속도를 하며 이것은 전혀 과학적이값은 아니다. 리눅스에 의하면 이 BogoMIPS값은 다음의 유용성이 있다.

1. 디버깅에 유용
 2. Computer cache와 turbo button 이 제대로 작동하는지 확인할수 있음.
- 실제로 BogoMIPS는 cpu 가 1 jiffy 동안 수행하는 empty loop의 횟수 값이다.

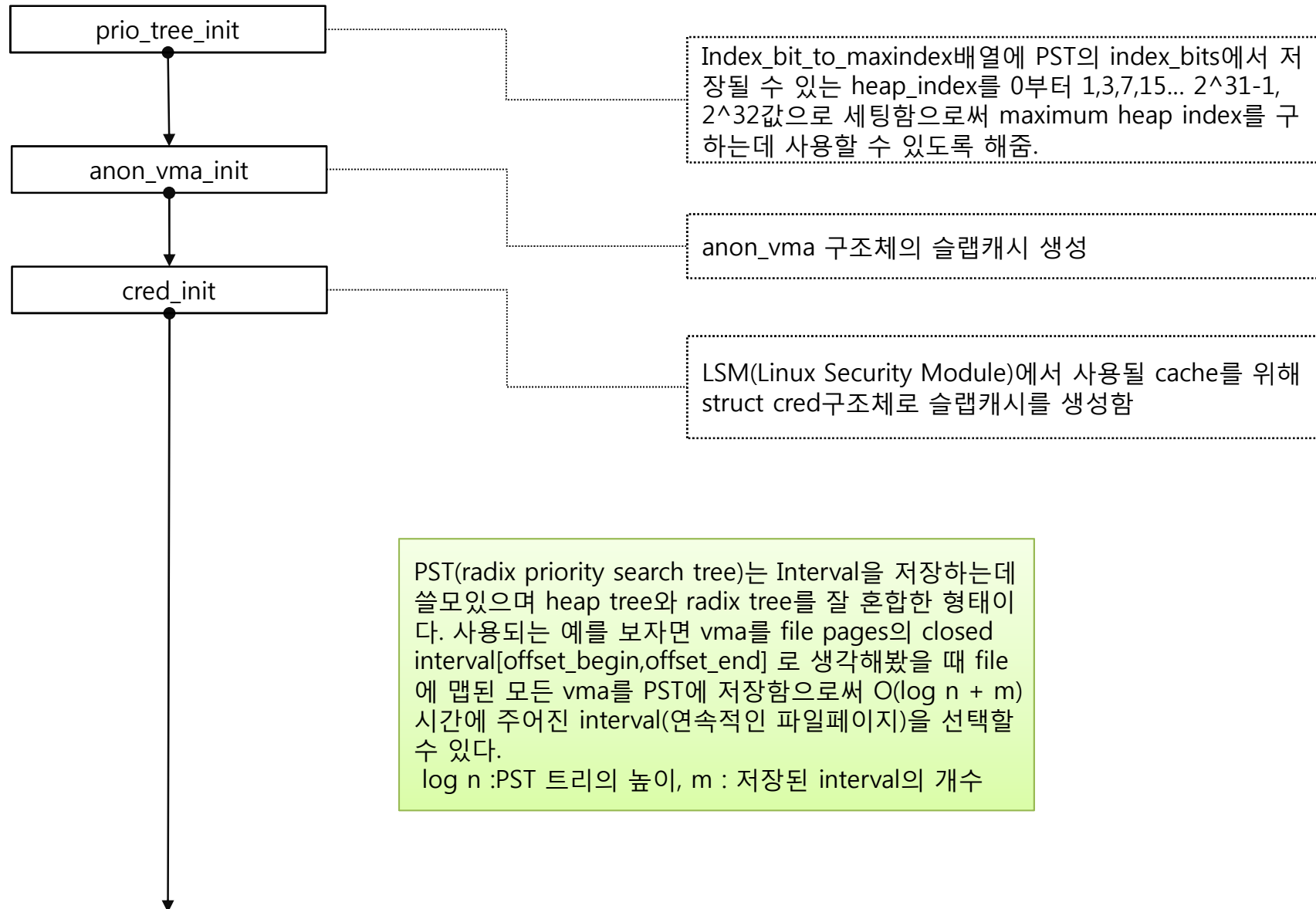
PID namespace

PID namespace는 태스크들의 집합을 만들도록 해준다. 즉 다른 namespace의 태스크들이 같은 ID를 가질 수 있다. 이 특징은 hosts사이의 이주를 위해 필수조건이다. PID namespace를 가짐으로써 프로세스는 PID값을 유지하면서 다른 host로 이동할 수 있다. 이특징이 없다면 호스트간 태스크 이주는 빈번히 실패할 것이다. 같은 ID를 가지는 프로세스가 존재할 수 있기 때문이다.

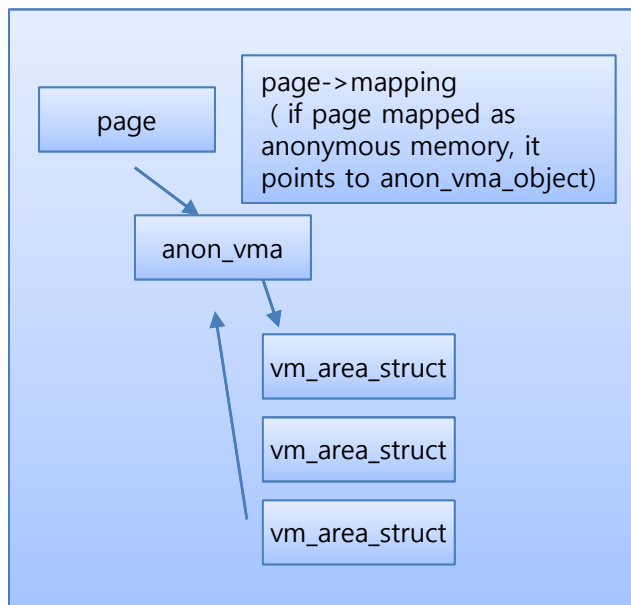
init/main.c



init/main.c



LSM(Linux Security Modules)는 리눅스 커널이 적재가능한 커널모듈로서 구현되어야 하는 다양한 access-control-model을 허용하기 위해 경량의 다목적 접근 제어 프레임워크로 구현되었다. 그 중에는 Security-Enhanced Linux(SELinux), Domain and Type Enforcement(DTE), Linux Intrusion Detection System(LIDS)가 있으며. LSM은 향상된 보안 정책을 커널 모듈로 적재가능하다.



Reverse-mapping virtual memory("rmap")

rmap은 swapping이 요구될때 커널이 메모리를 free 하는 것을 쉽게 만드는 목적이였다. 이를 위해 reverse pointer chain에서 각 포인터는 페이지를 참조하는 페이지테이블을 가리킨다. rmap chain에 의해 커널은 빠르게 주어진 페이지의 모든 매핑을 찾을수 있고 unmap 이후 page를 swap out할수 있다.

anonymous page를 참조하는 다수의 VMA를 anon_vma를 통해 연결시키는데 이것은 page->mapping 포인터가 anon_vma를 가리킴으로써 커널은 list를 순회하여 연관된 VMA structure를 찾을 수 있다.

init/main.c

fork_init

- 1) (struct task_struct) 의 슬랩을 생성
- 2) 전역 변수인 `max_threads` 의 값을 정한다. (생성할 수 있는 스레드 최대 갯수)
- 3) 현재 부팅을 주도하고있는 `init_task`가 생성할 수 있는 스레드 갯수를 정해준다. (`init_task.signal.rlim[RLIMIT_NPROC]` 값 설정)
- 4) 현재 부팅을 주도하고있는 `init_task`의 최대 pending signal 갯수를 정해준다. (`init_task.signal.rlim[RLIMIT_SIGPENDING]` 값 설정)

kernel/fork.c

int max_thread

arch/arm/kernel/init_task.c

struct task_struct init_task

struct task_struct

...
struct signal_struct *signal
...

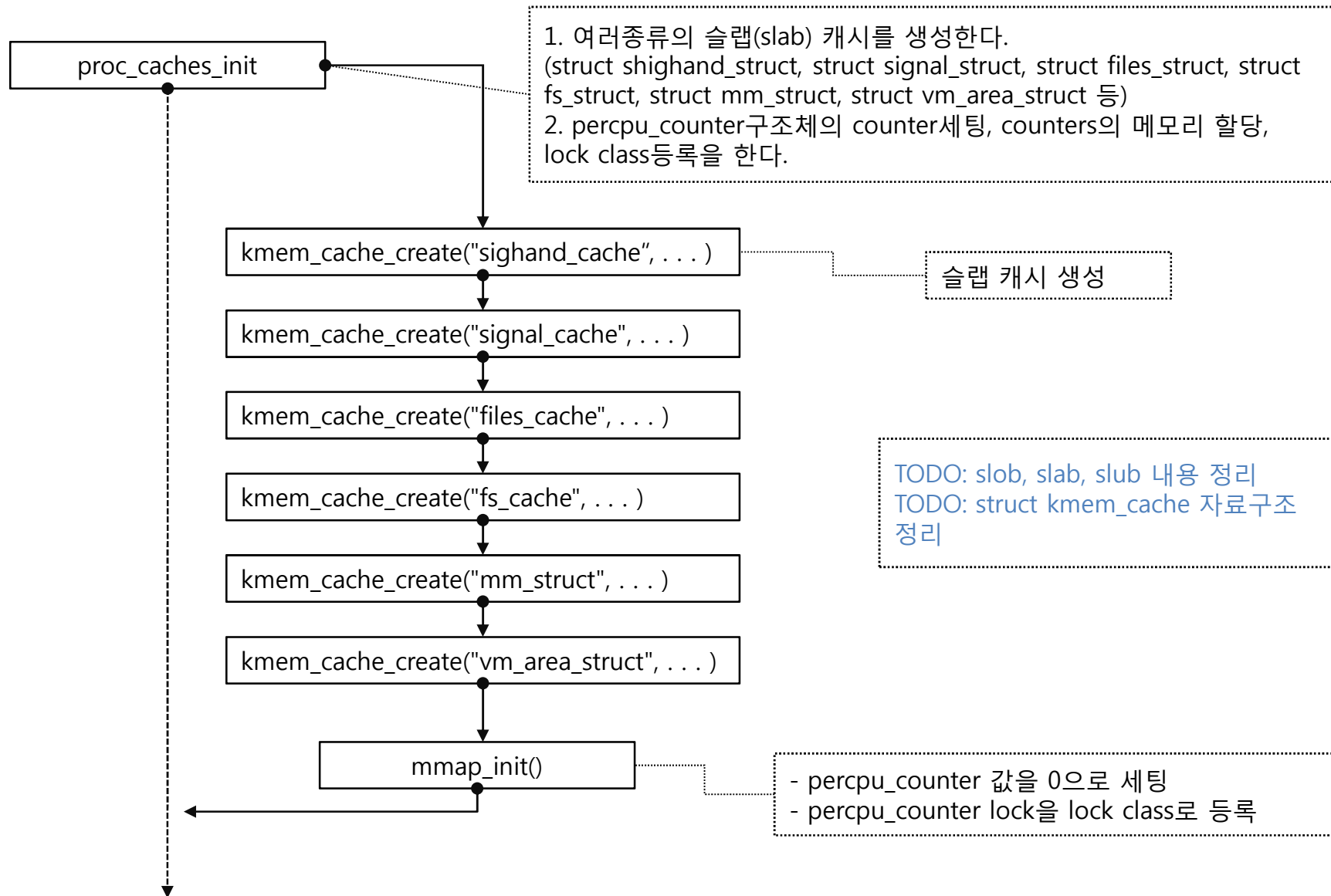
struct task_struct

...
struct rlimit rlim[RLIM_NLIMITS]
...

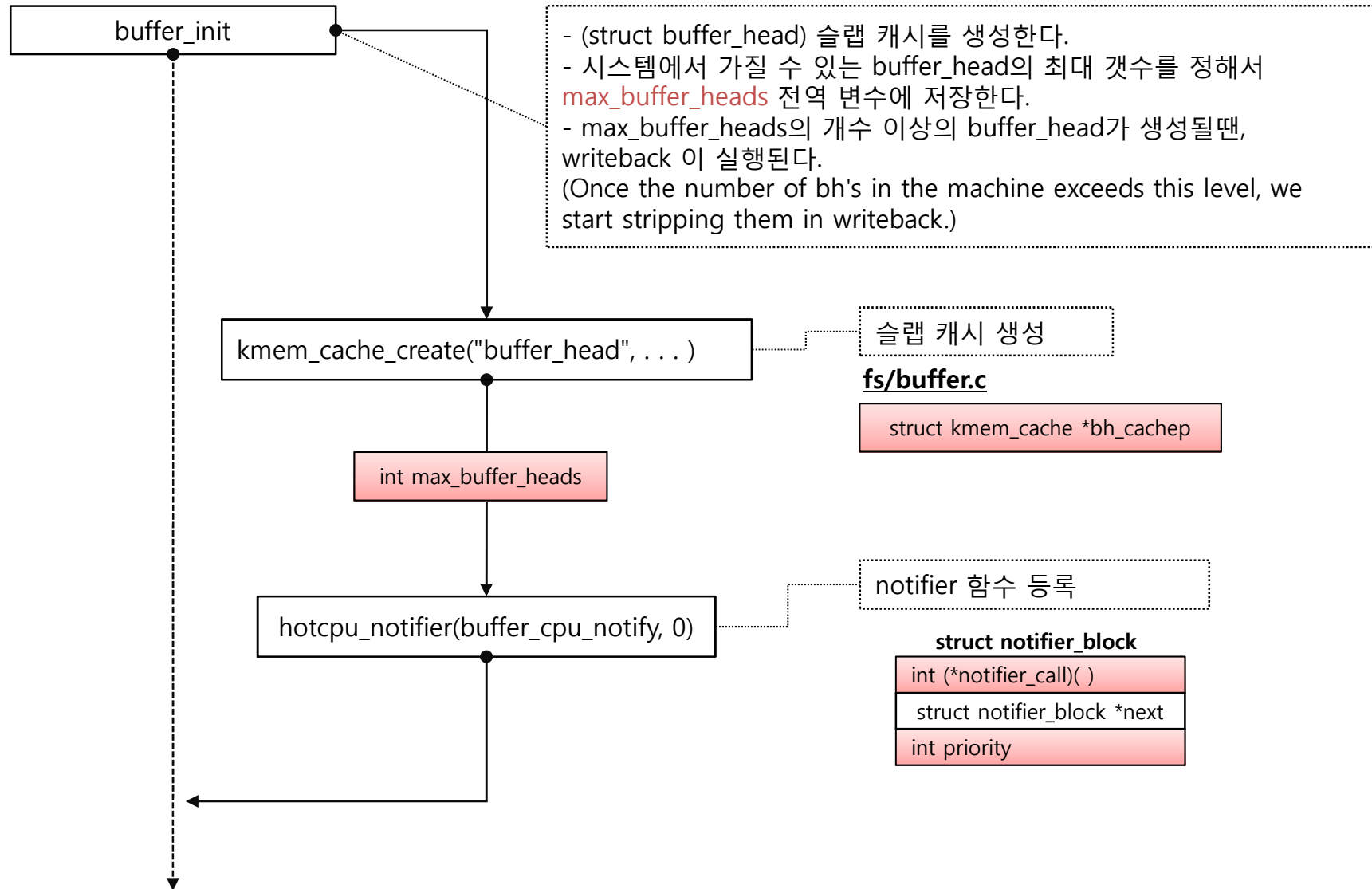
Include/asm-generic/resource.h

```
...  
#define RLIMIT_NPROC      6 /* max number of processes */  
#define RLIMIT_SIGPENDING 11 /* max number of pending signals */  
#define RLIM_NLIMITS     16 // maximum file locks held  
...
```

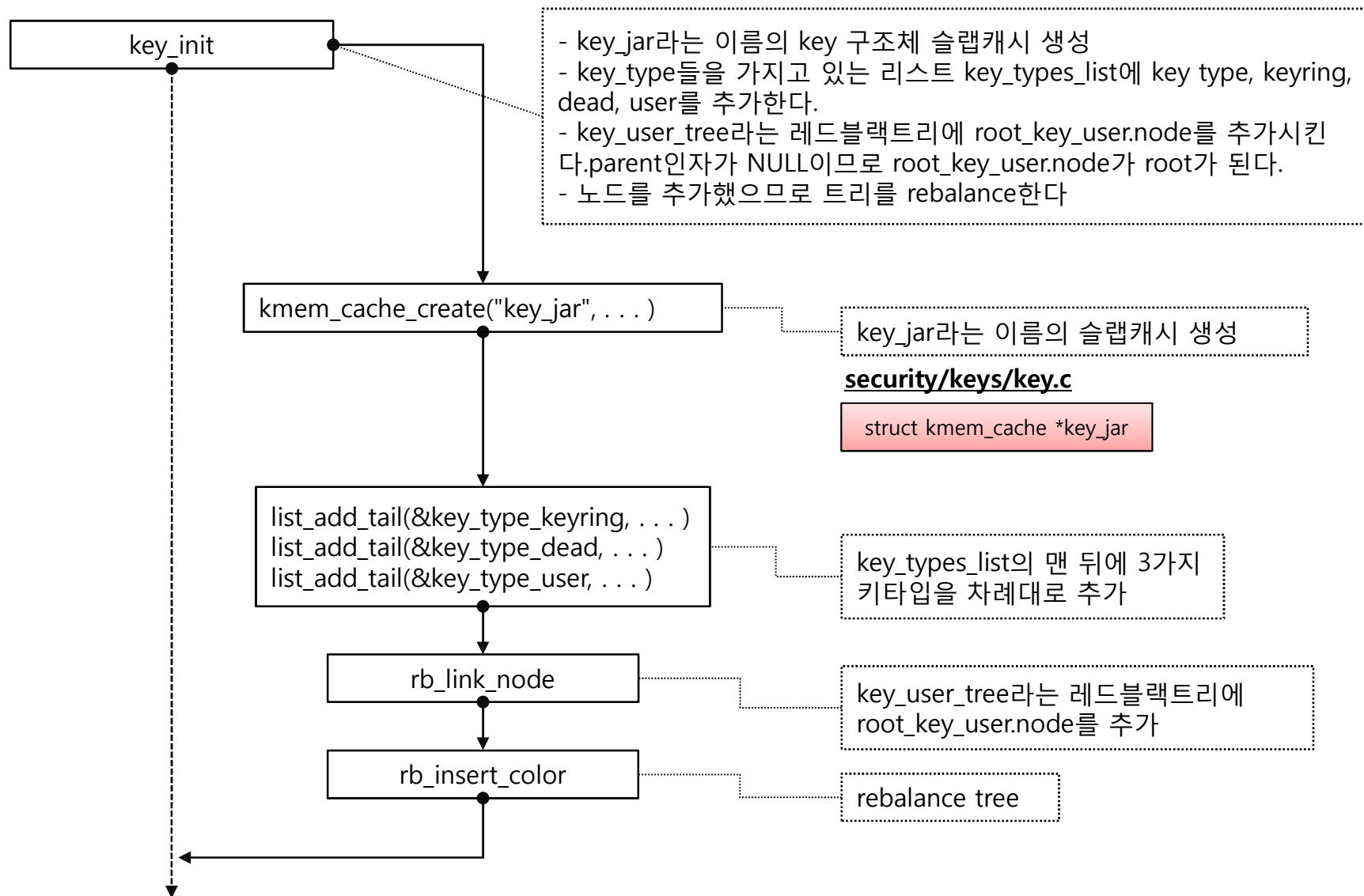
init/main.c



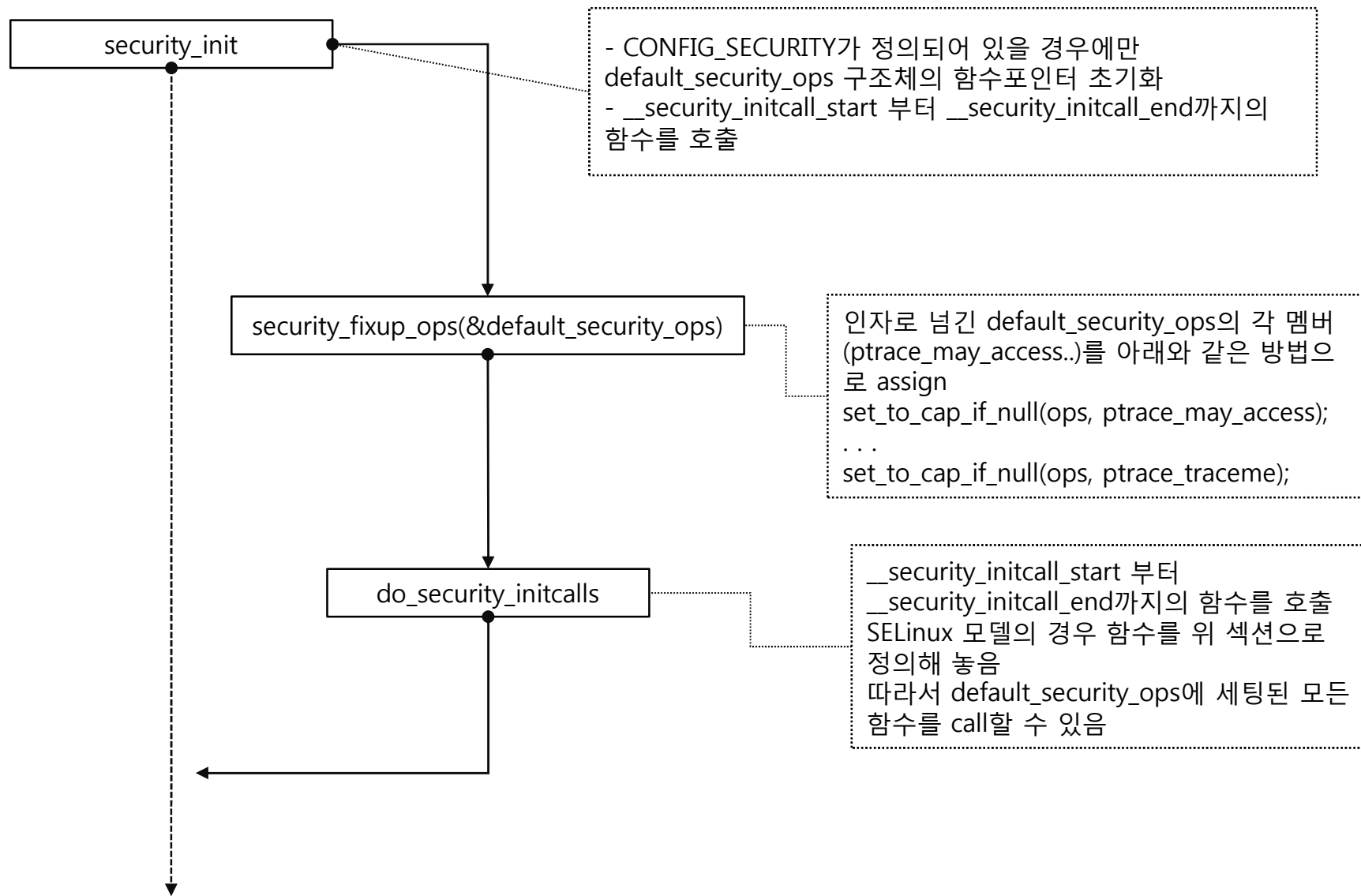
init/main.c



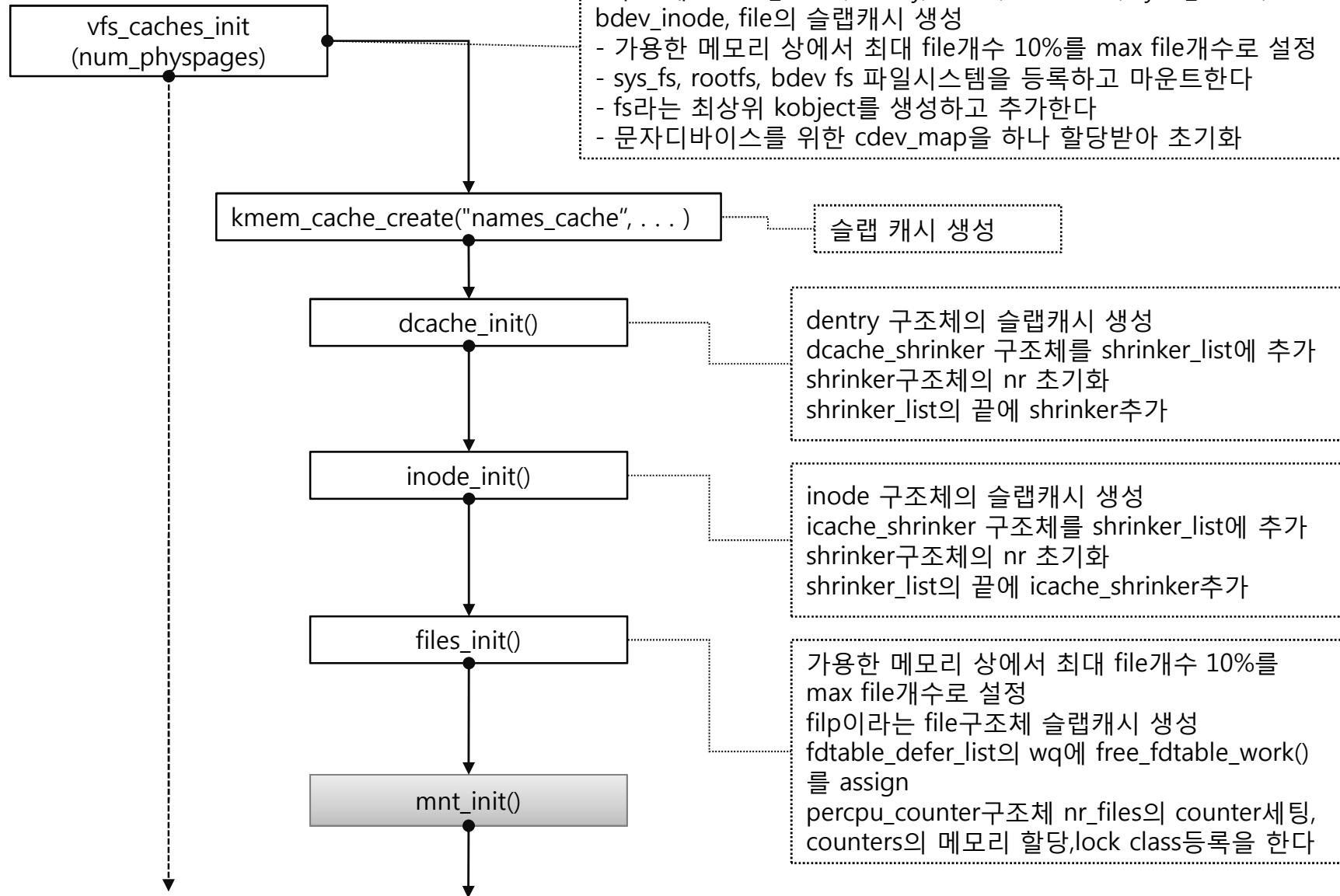
init/main.c



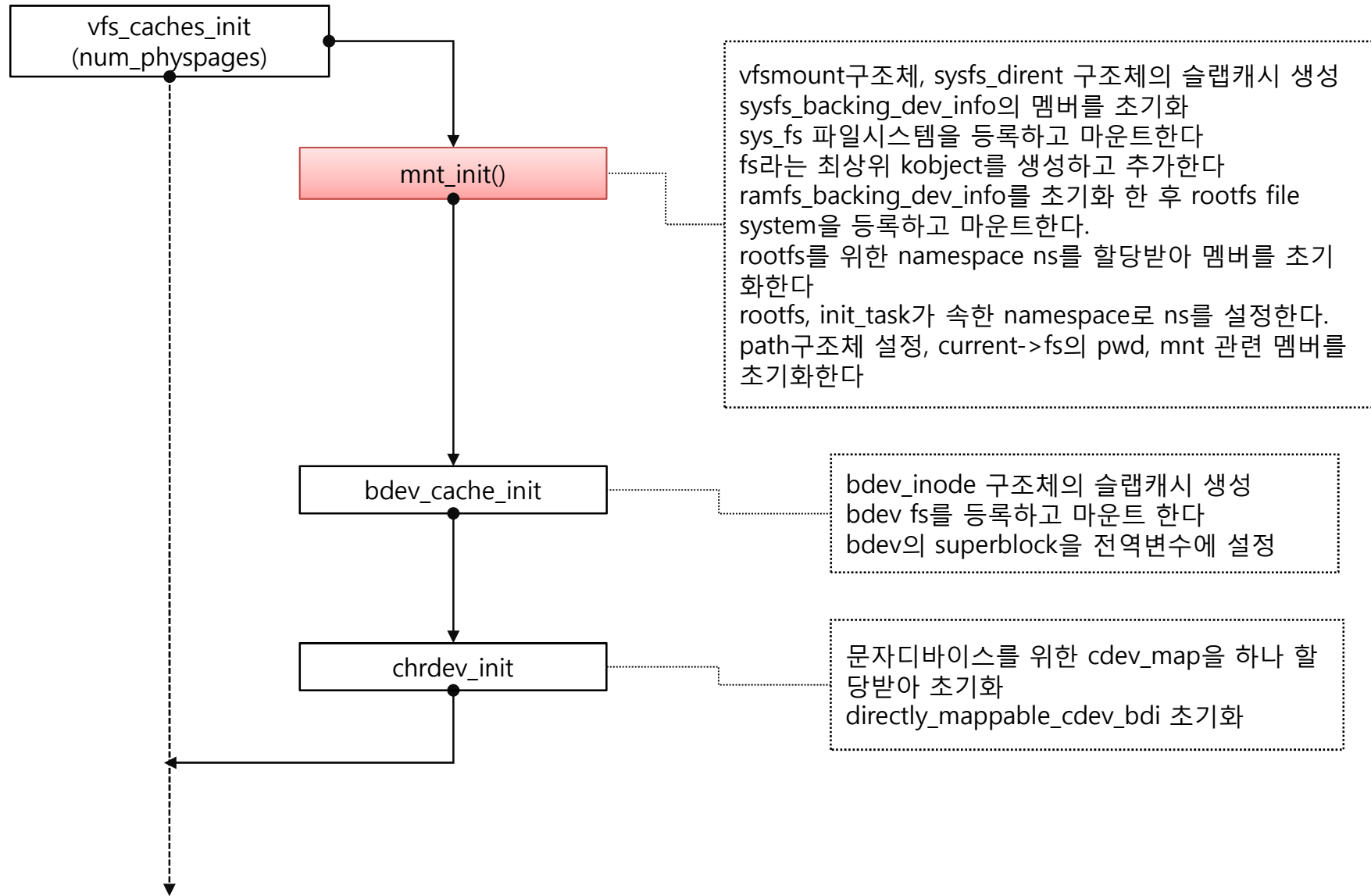
init/main.c



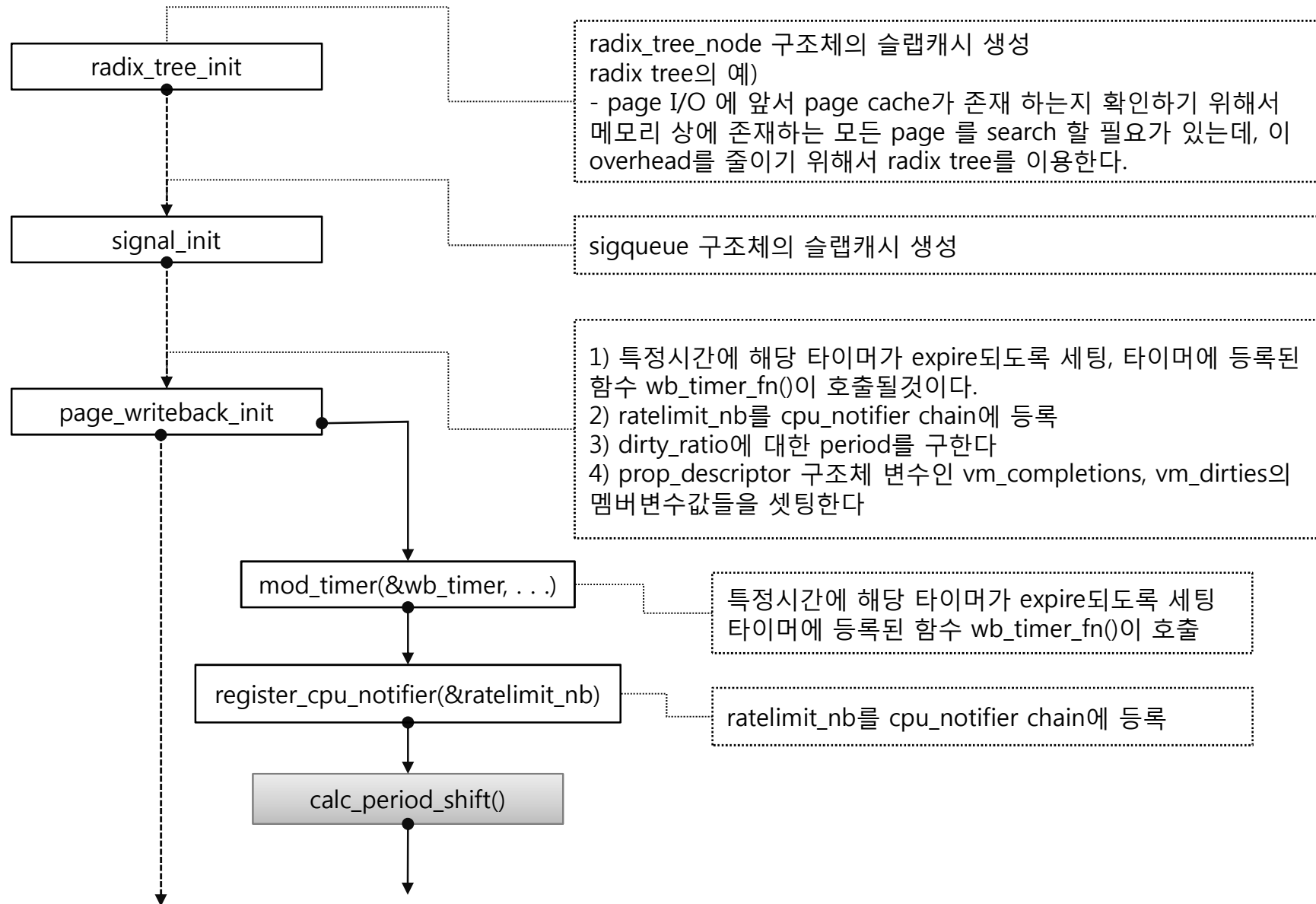
init/main.c



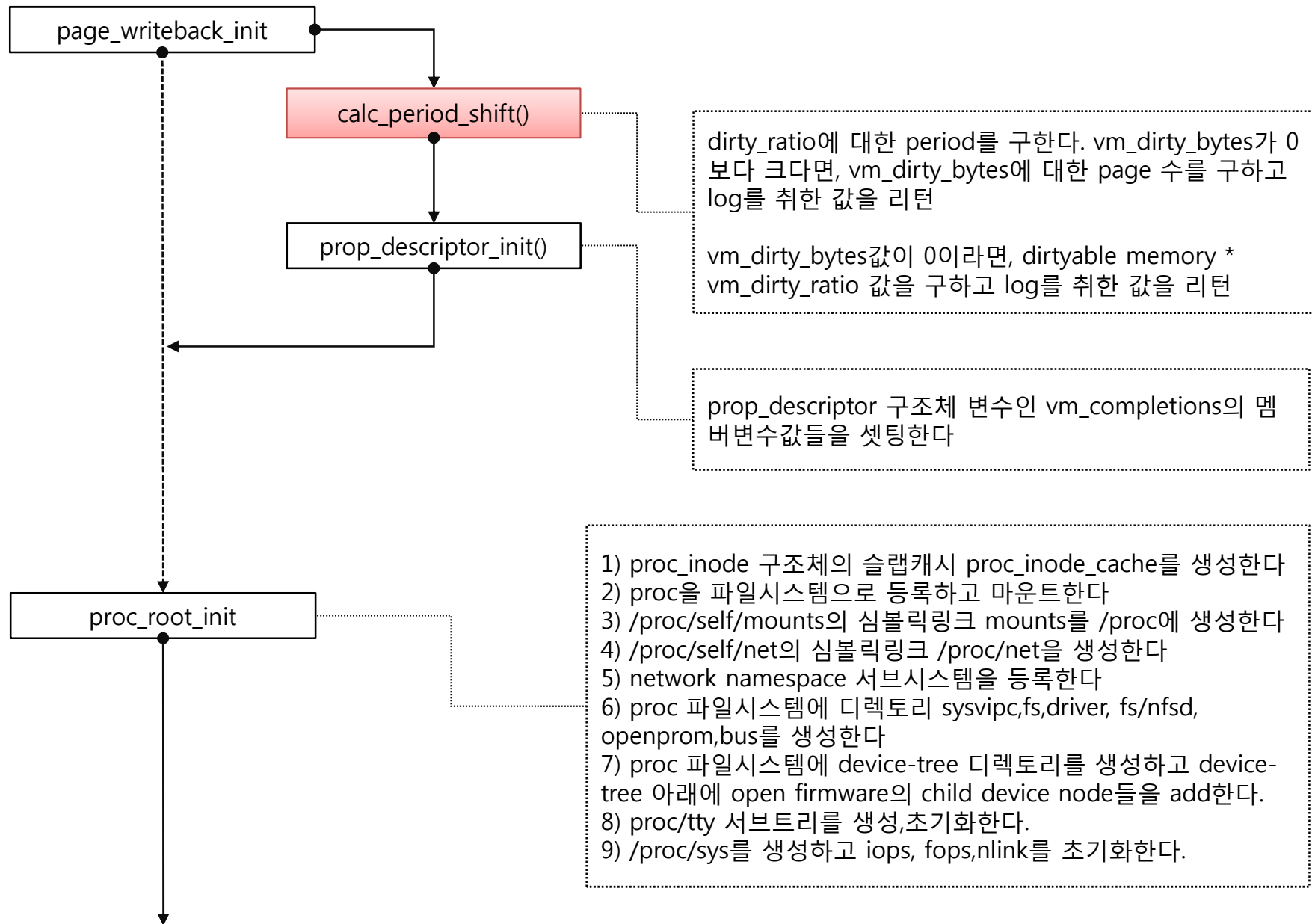
init/main.c



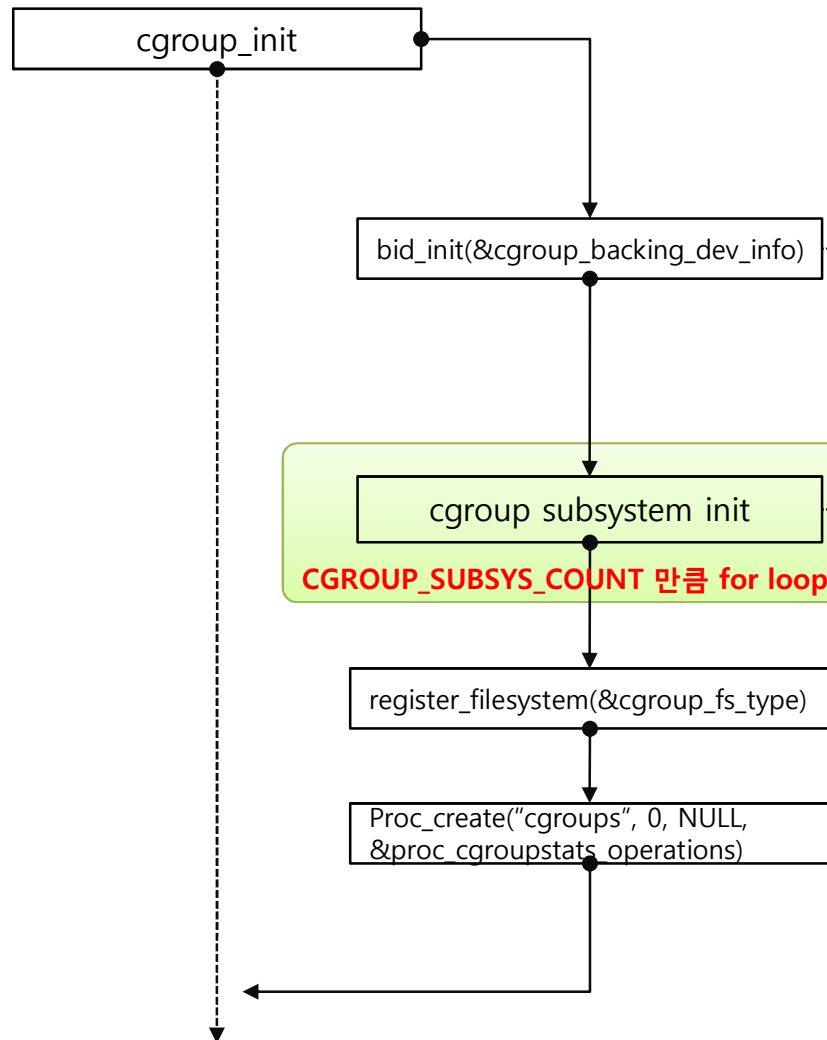
init/main.c



init/main.c



init/main.c



cgroup init() 함수의 주요 Operation

1. `cgroup_backing_dev_info`의 멤버를 초기화
2. `cgroup`의 subsystem 전부 initialize
3. `cgroup` subsystem 의 해시테이블에 `init_css_set`을 추가
4. `cgroup` 파일시스템을 등록한다
5. `proc`파일시스템에 `cgroups` 엔트리 생성

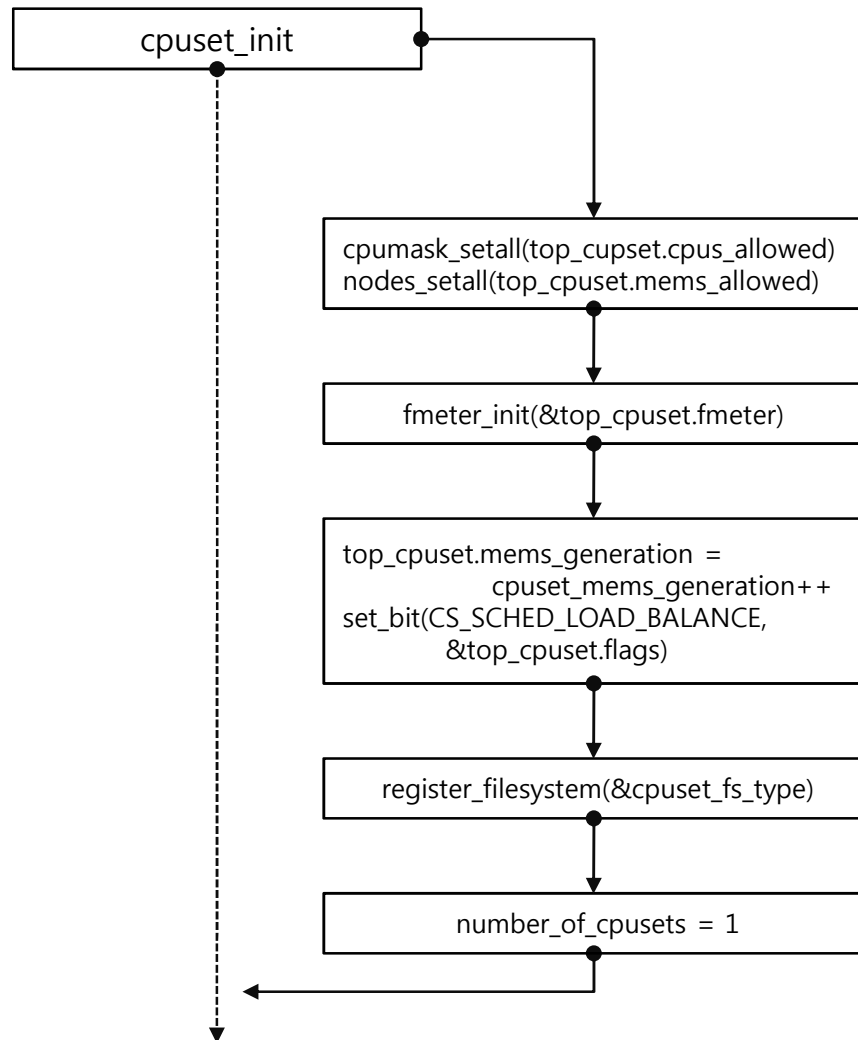
1. `cgroup_backing_dev_info`의 멤버를 초기화
2. `percpu_counter` 구조체인 `bdi_stat[]`, `completions`의 `counter`를 0으로 세팅, `counters`의 메모리 할당, `lock class`등록을 한다

`cgroup`의 subsystem (`cpuset`) (`debug`) (`ns`) (`cpu_cgroup`) (`cpuacct`) (`mem_cgroup`) (`devices`) (`freezer`) (`net_cls`) 모두 initialize

`cgroup` 파일시스템을 등록한다.

`proc` 파일시스템에 `cgroups` 엔트리 생성

init/main.c



cpuset init() 함수의 주요 Operation

1. 전역변수 `top_cpuset`의 `cpus_allowed`, `mems_allowed`를 각각 cpu개수만큼, `MAX_NUMNODES`만큼 set
2. `top_cpuset`구조체의 `fmeter` 구조체를 초기화
3. `cpuset fs`를 파일시스템으로 등록한다
4. `cpumask`인 `cpus_attach`에 메모리를 할당
5. 현재 시스템에 있는 `cpuset`개수를 세팅

1. `top_cpuset`의 `cpus_allowed`, `mems_allowed`를
2. 각각 cpu개수만큼, `MAX_NUMNODES`만큼 set

`top_cpuset`구조체의 `fmeter` 구조체를 초기화 `fmeter`는 frequency meter인데 어떤 이벤트가 얼마나 빠르게 발생하는지에 대한 정보를 가지고 있음

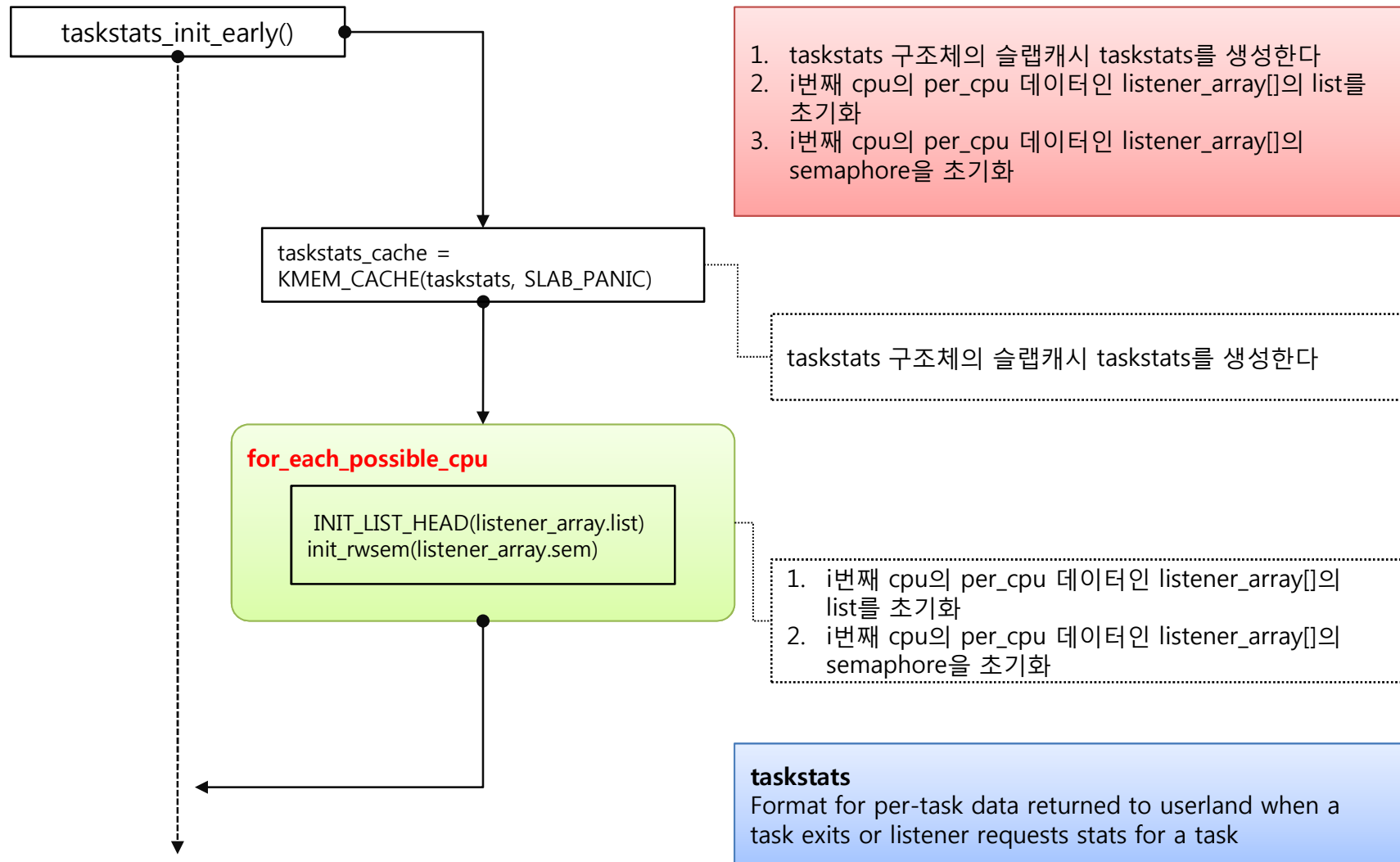
어떠한 `cpuset`이 `mems_allowed`의 값을 변경하면 `mems_generation`의 값을 증가시킨다

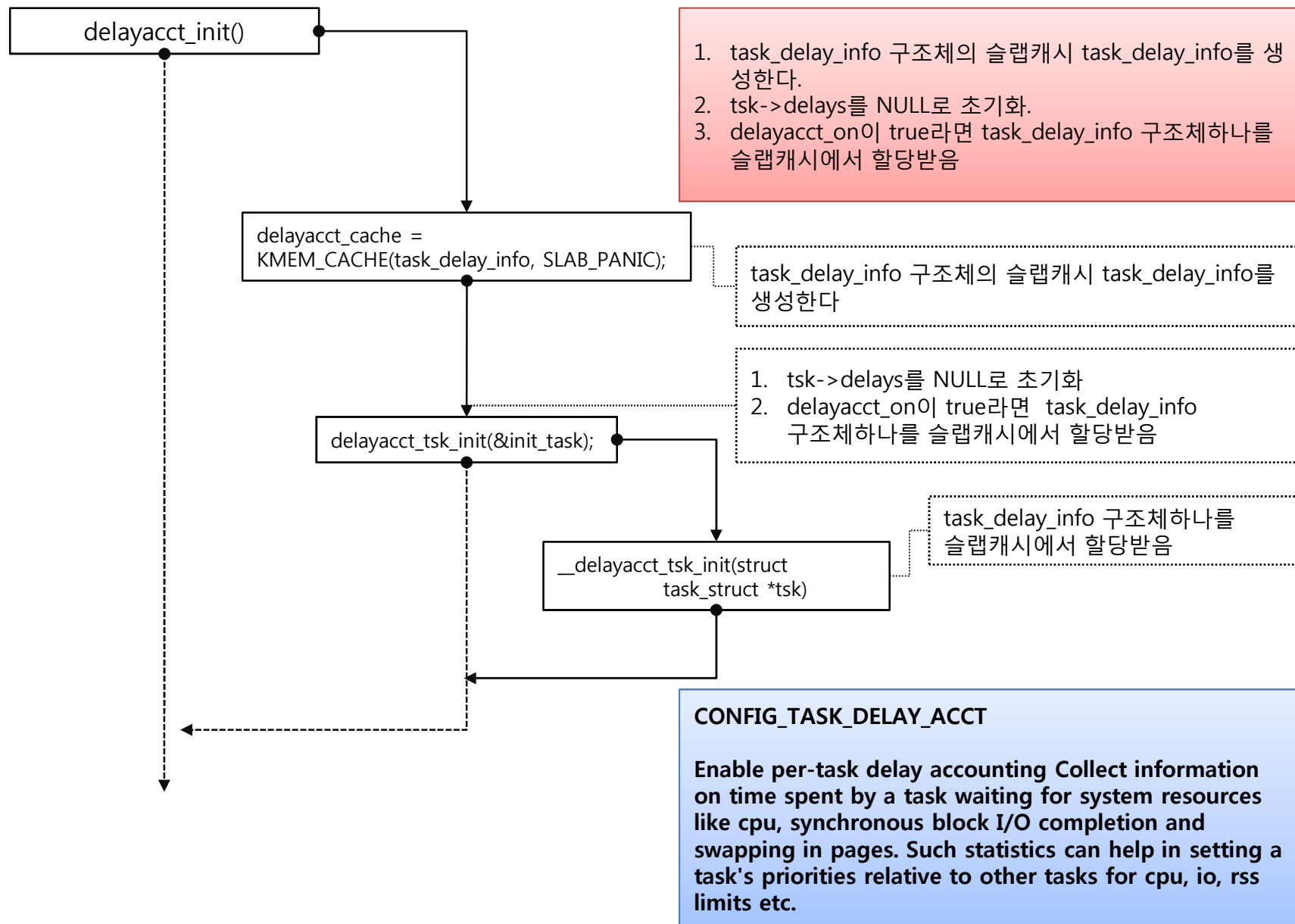
`cpuset fs`를 파일시스템으로 등록한다.

현재 시스템에 있는 `cpuset`개수를 세팅

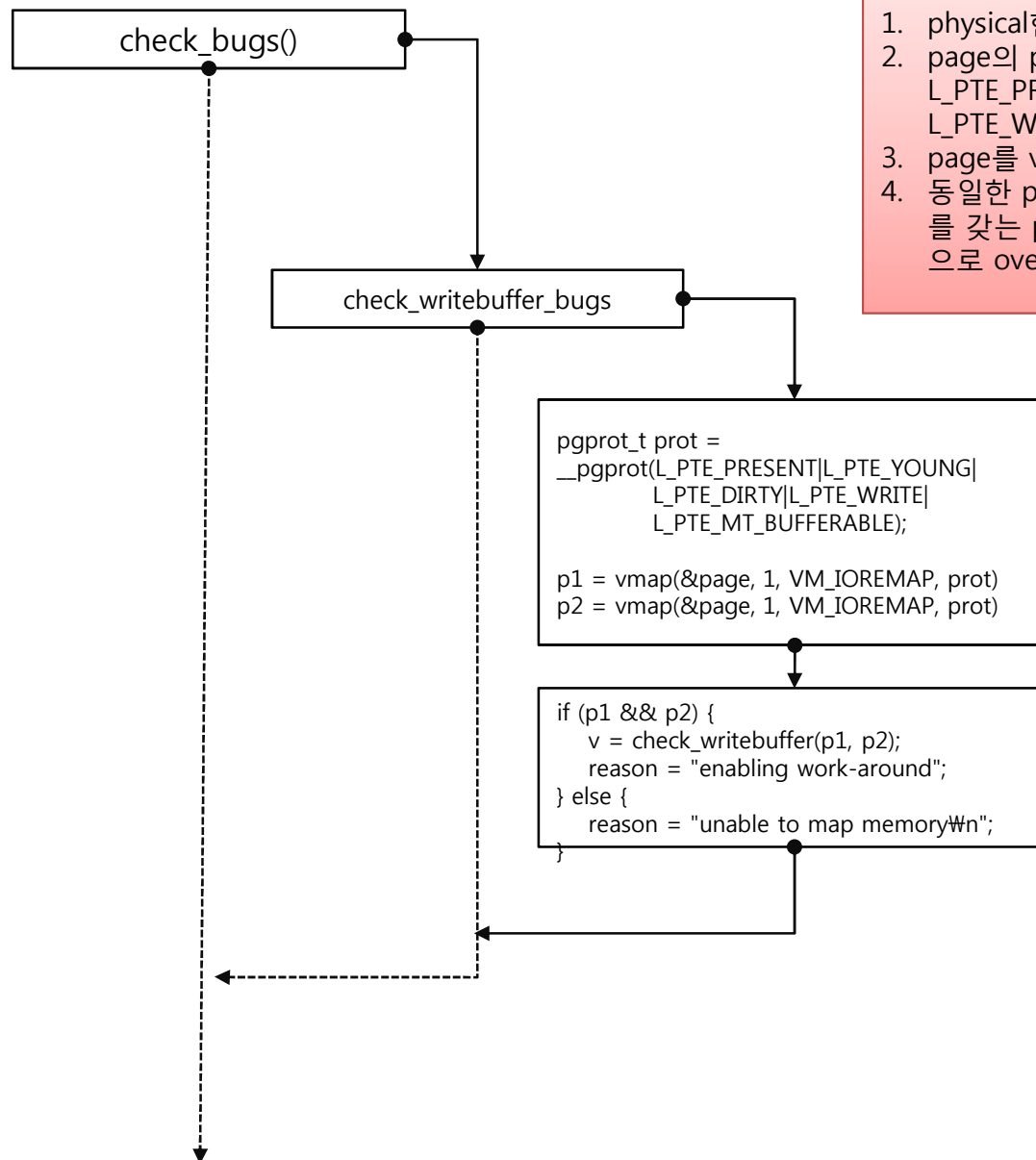
init/main.c

taskstats init early() 함수의 주요 Operation





init/main.c



check_bugs() 함수의 주요 Operation

1. physical한 page 한개 할당받는다
2. page의 protection을 `L_PTE_PRESENT|L_PTE_YOUNG|L_PTE_DIRTY|L_PTE_WRITE| L_PTE_MT_BUFFERABLE`로 설정한다
3. page를 virtual address `p1,p2`로 매핑한다
4. 동일한 physical page에 대한 서로 다른 virtual address를 갖는 `p1, p2`에 writing test. `p1`에 쓴값이 `p2`에 쓴값으로 over write된다면 테스트 성공

1. physical한 page 한개 할당 받는다
2. page의 protection을 `L_PTE_PRESENT | L_PTE_YOUNG | L_PTE_DIRTY | L_PTE_WRITE| L_PTE_MT_BUFFERABLE`로 설정한다
3. page를 virtual address `p1, p2`로 매핑한다

동일한 physical page에 대한 서로 다른 virtual address를 갖는 `p1, p2`에 writing test를 하는 함수 `p1`에 쓴값이 `p2`에 쓴값으로 over write된다면 테스트 성공

init/main.c

acpi_early_init()

ftrace_init()

count = __stop_mcount_loc -
__start_mcount_loc

ftrace_dyn_table_alloc(count)

ftrace_convert_nops(NULL,
__start_mcount_loc,
__stop_mcount_loc)

1. ACPI (Advanced Configuration and Power Interface) Support
2. for X86 ref page → <http://www.spinics.net/lists/arm-kernel/msg01205.html>

no-op으로 시작하는 함수의 주소 갯수

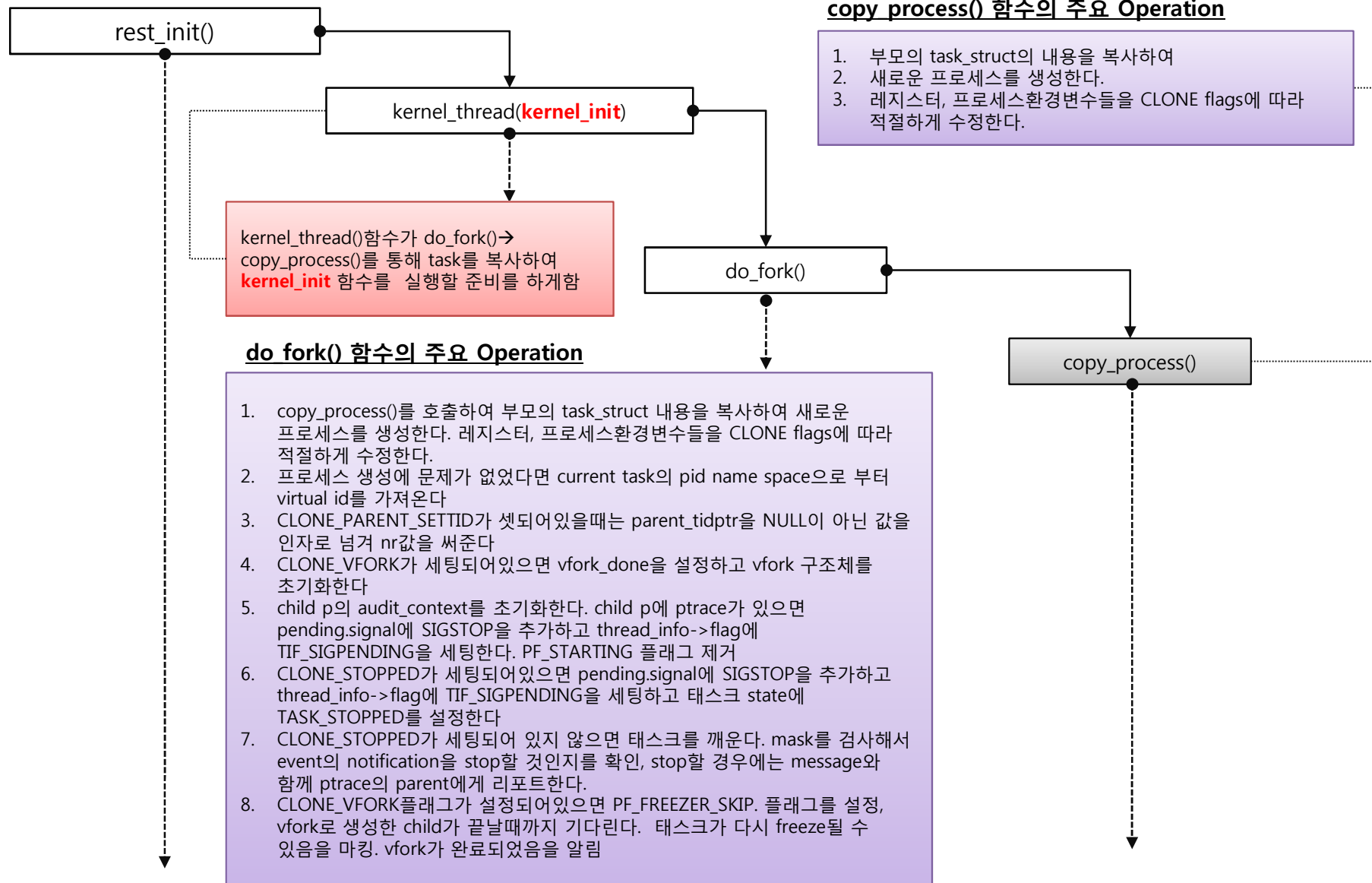
인자로 넘어온 num_to_init은 no-op으로 시작하는 함수의 총 주소 개수 num_to_init이 몇페이지에 들어갈 수 있는지를 계산하여 페이지를 할당 받음

1. ftrace를 위한 entry 개수만큼 루프를 돌면서 dyn_ftrace page로부터 record를 담은 주소를 리턴받아 멤버세팅
2. ftrace의 entry인 dyn_ftrace의 ip는 MCOUNT_ADDR을 가리키고 있음 ip를 no-op으로 변경한다
3. no-op으로 바꾸는 시간을 측정
4. 변경된 횟수를 저장

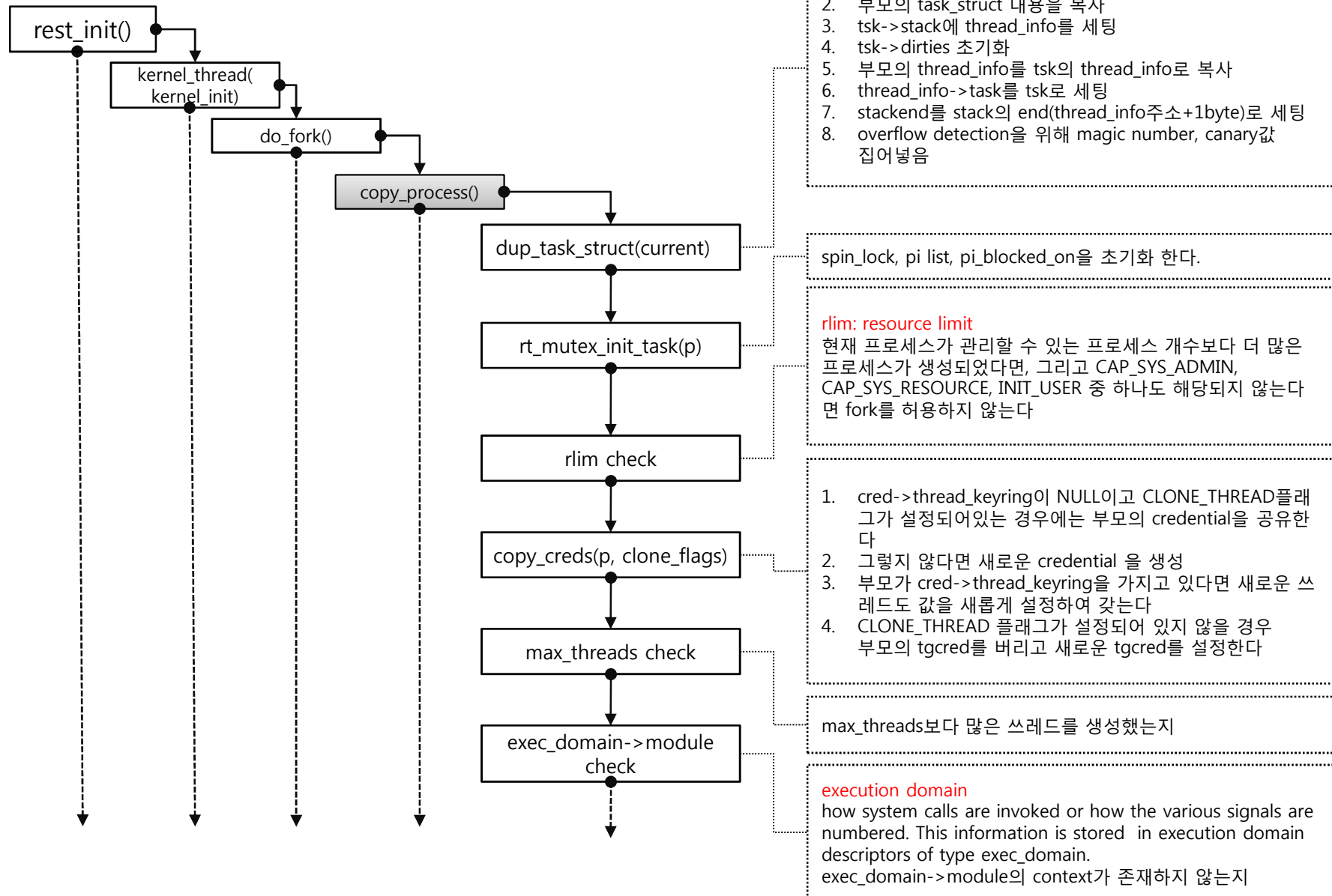
ftrace(function trace)

컴파일 타임에 모든 커널함수의 앞에 5byte No-op instruction을 삽입 ftrace가 enable 되면 no-op instruction이 ftrace에 의해 함수를 trace 하는 코드로 교체된다.

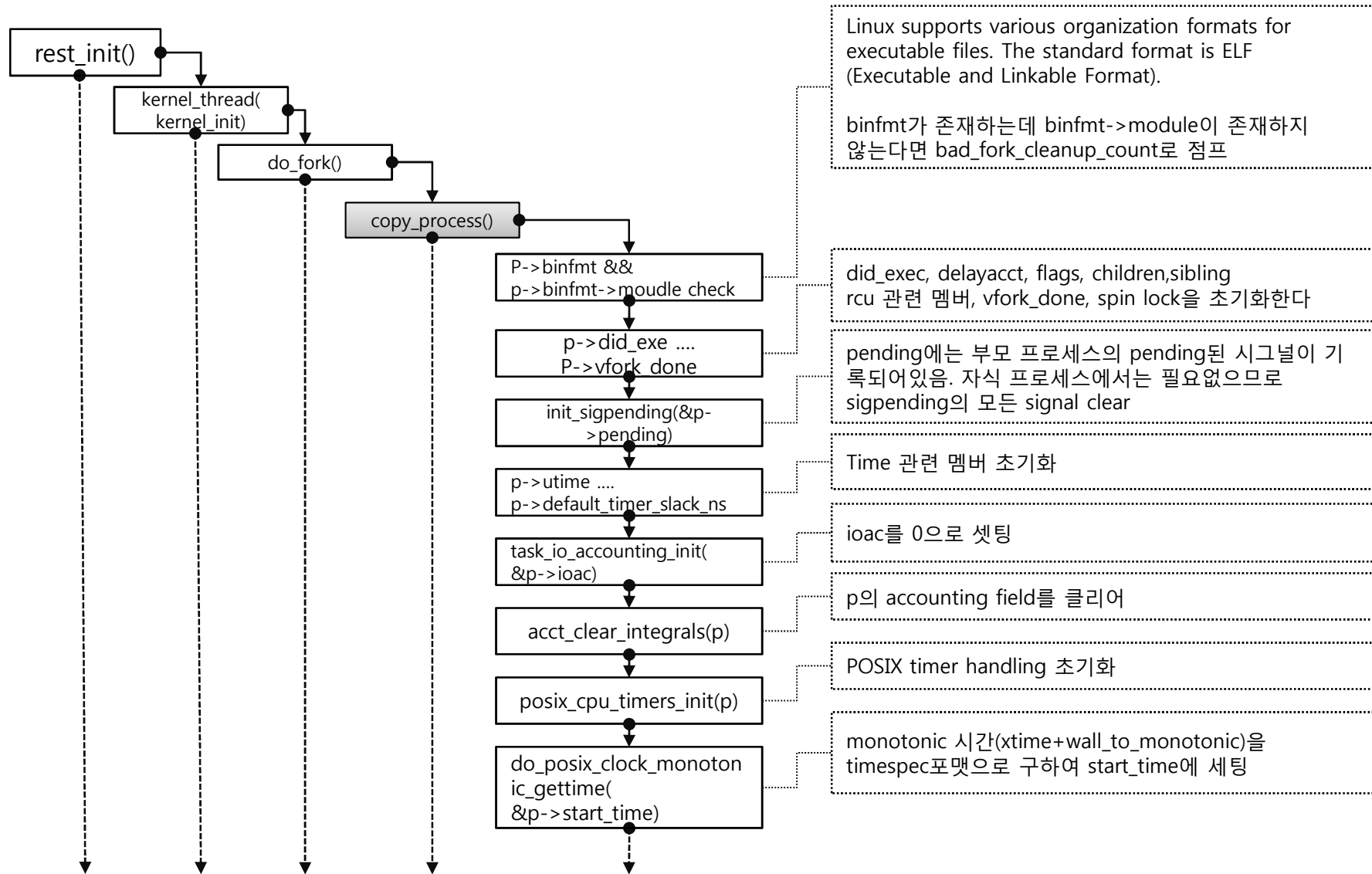
init/main.c



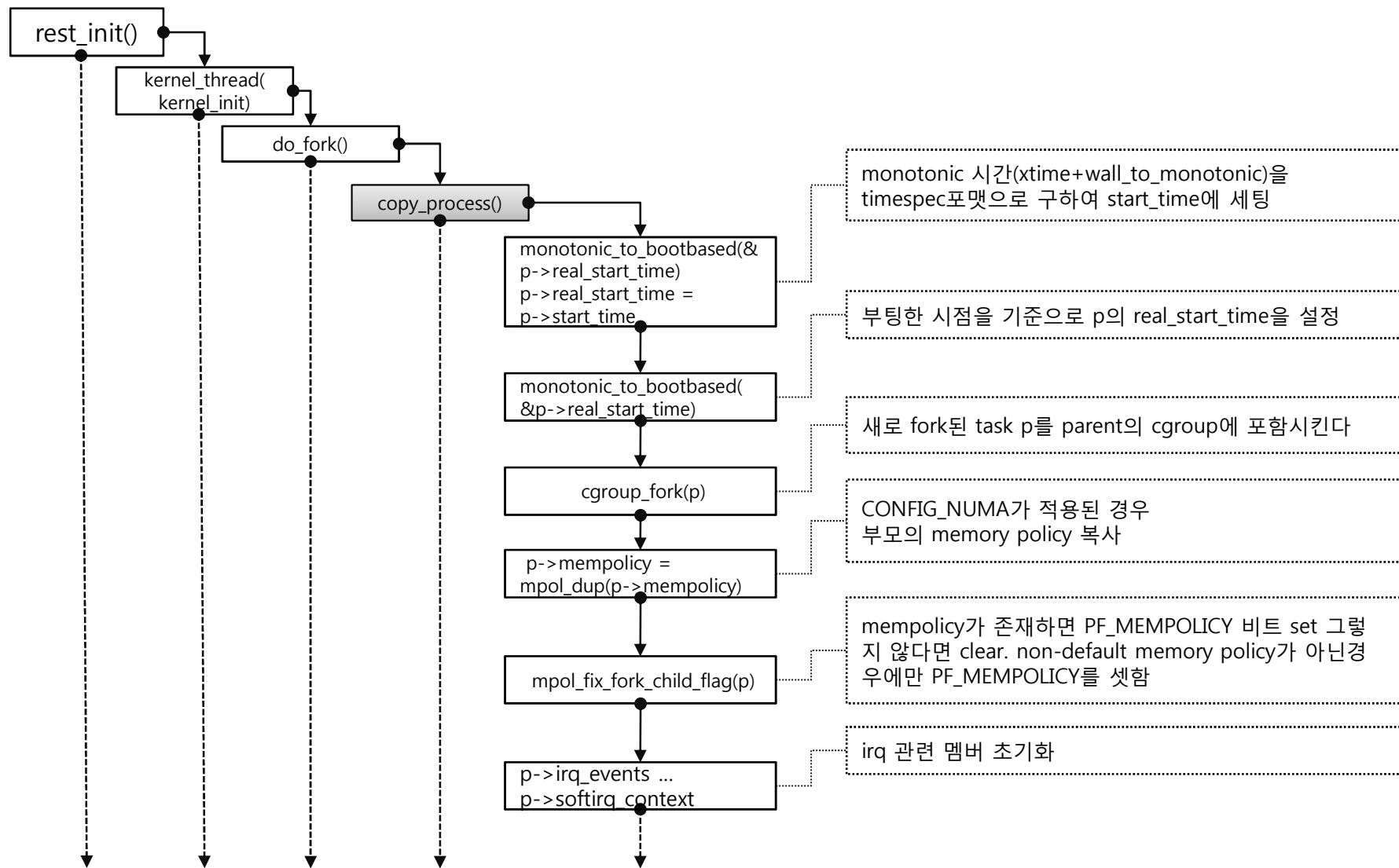
init/main.c



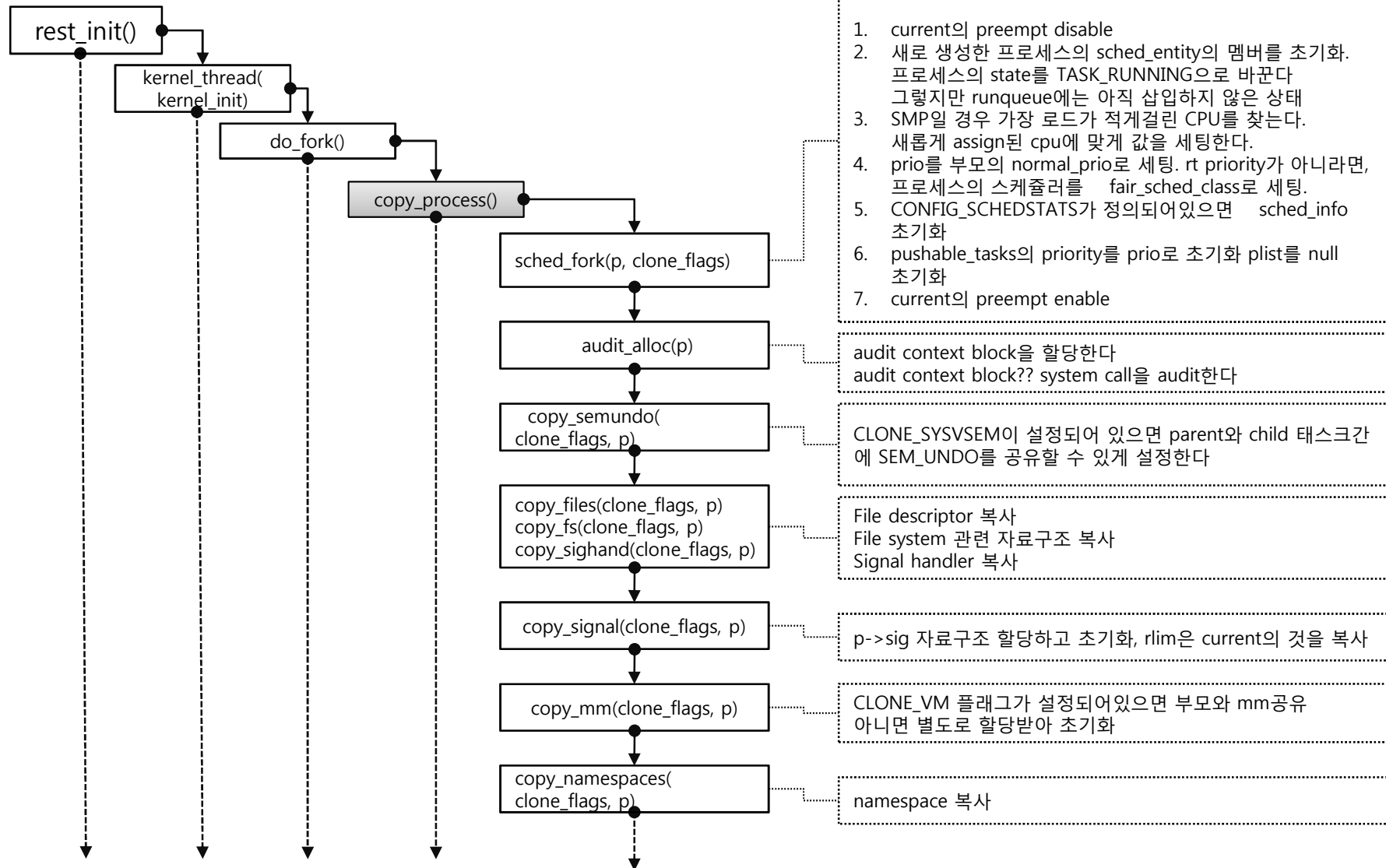
init/main.c



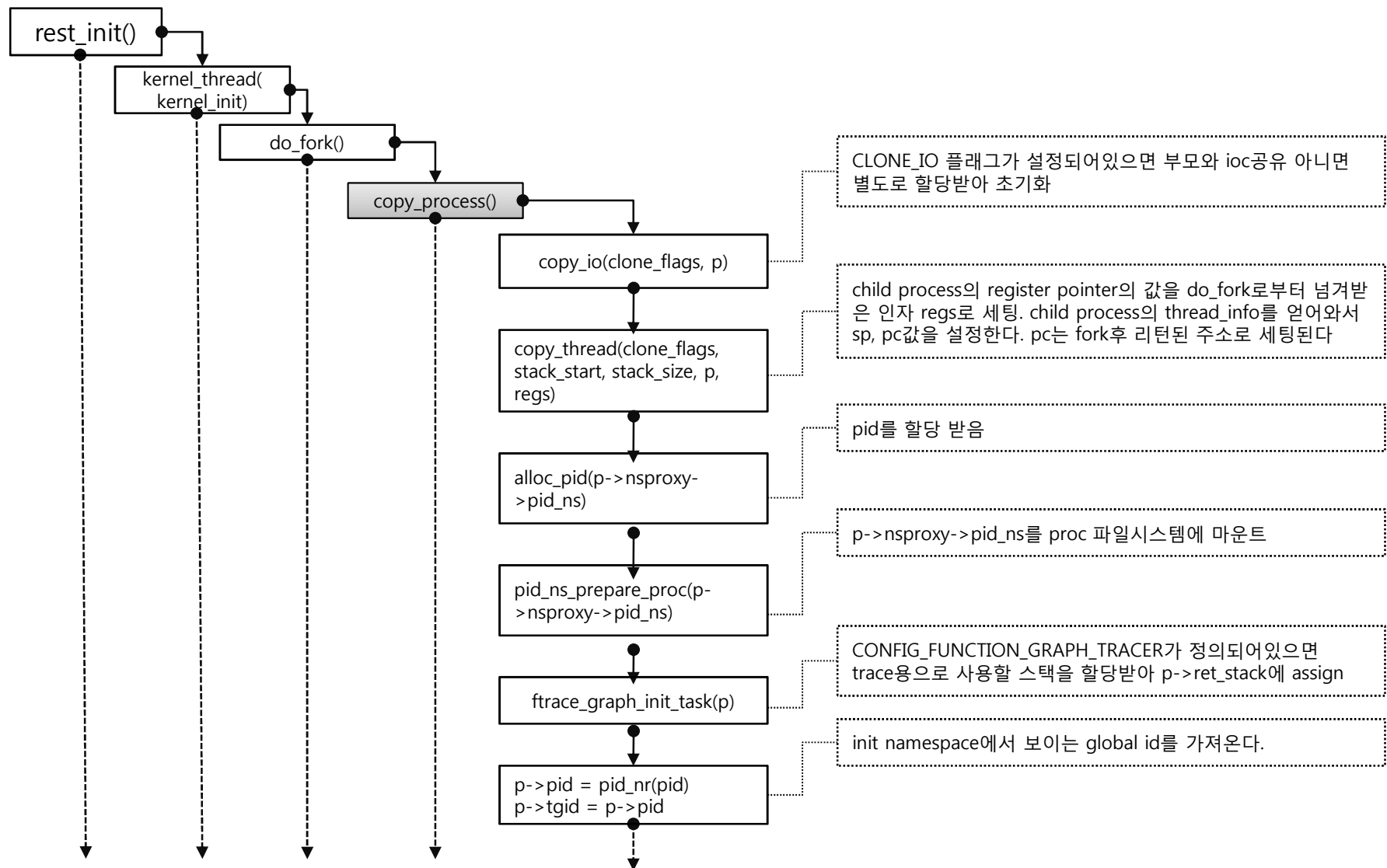
init/main.c



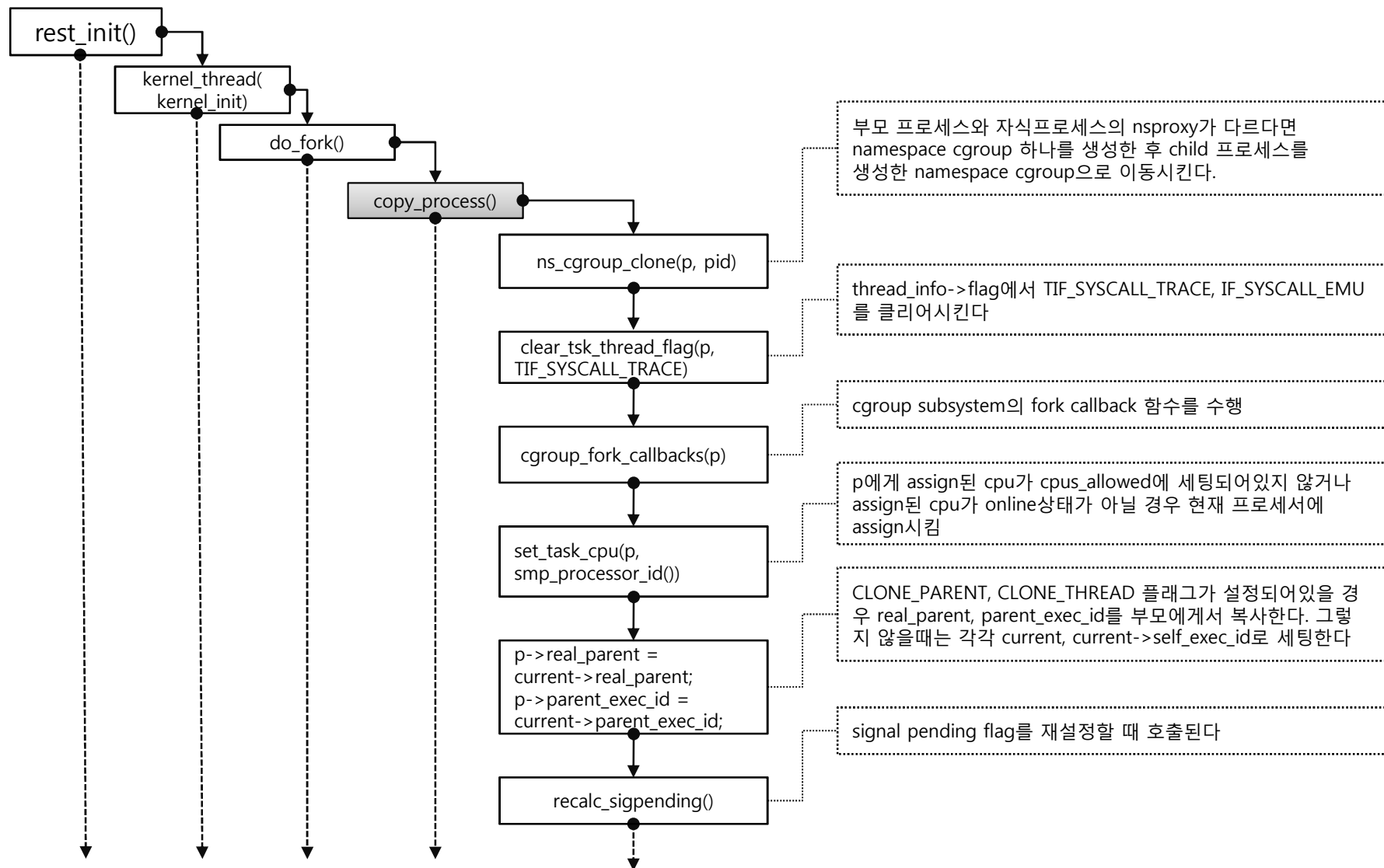
init/main.c



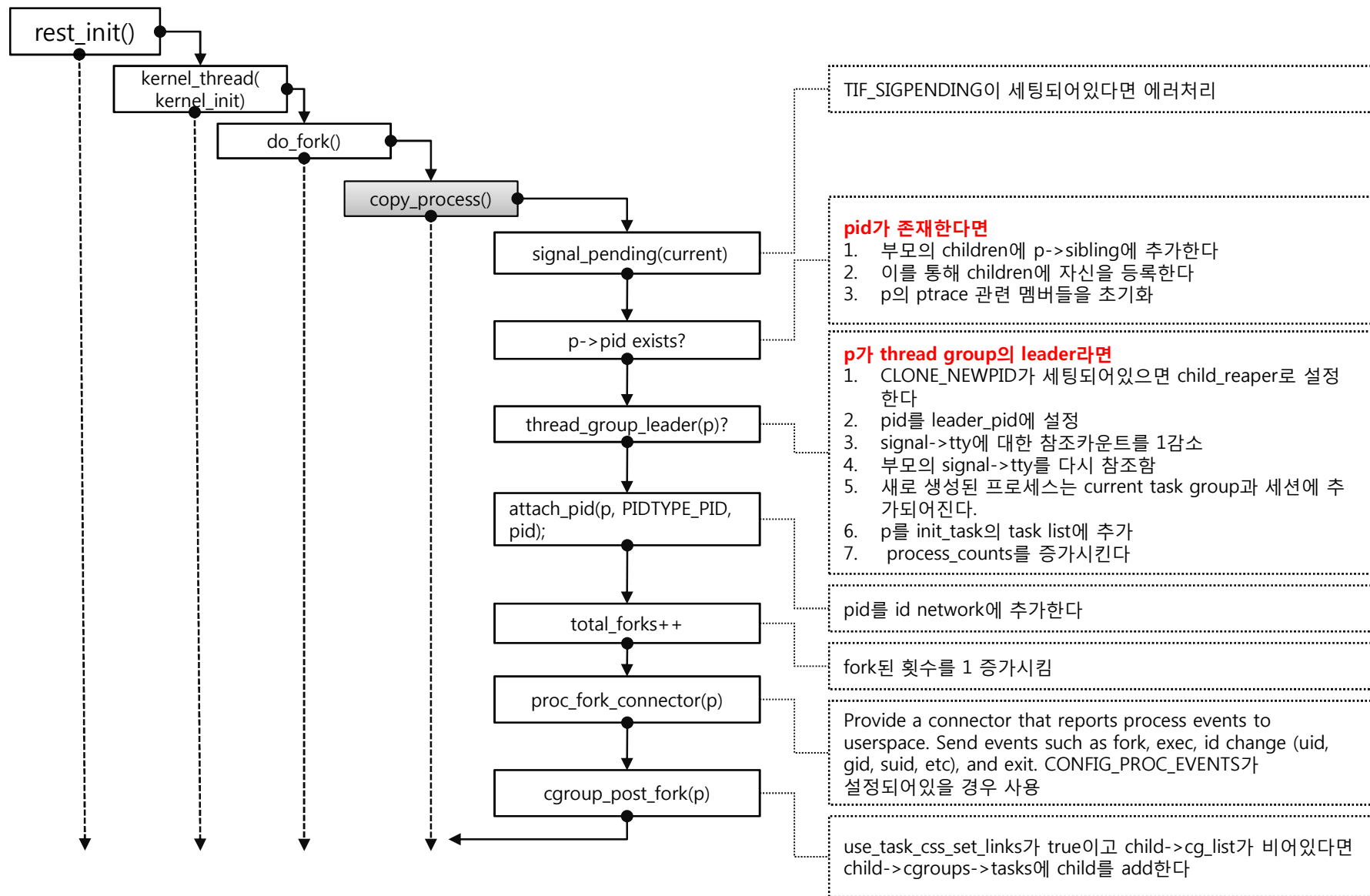
init/main.c



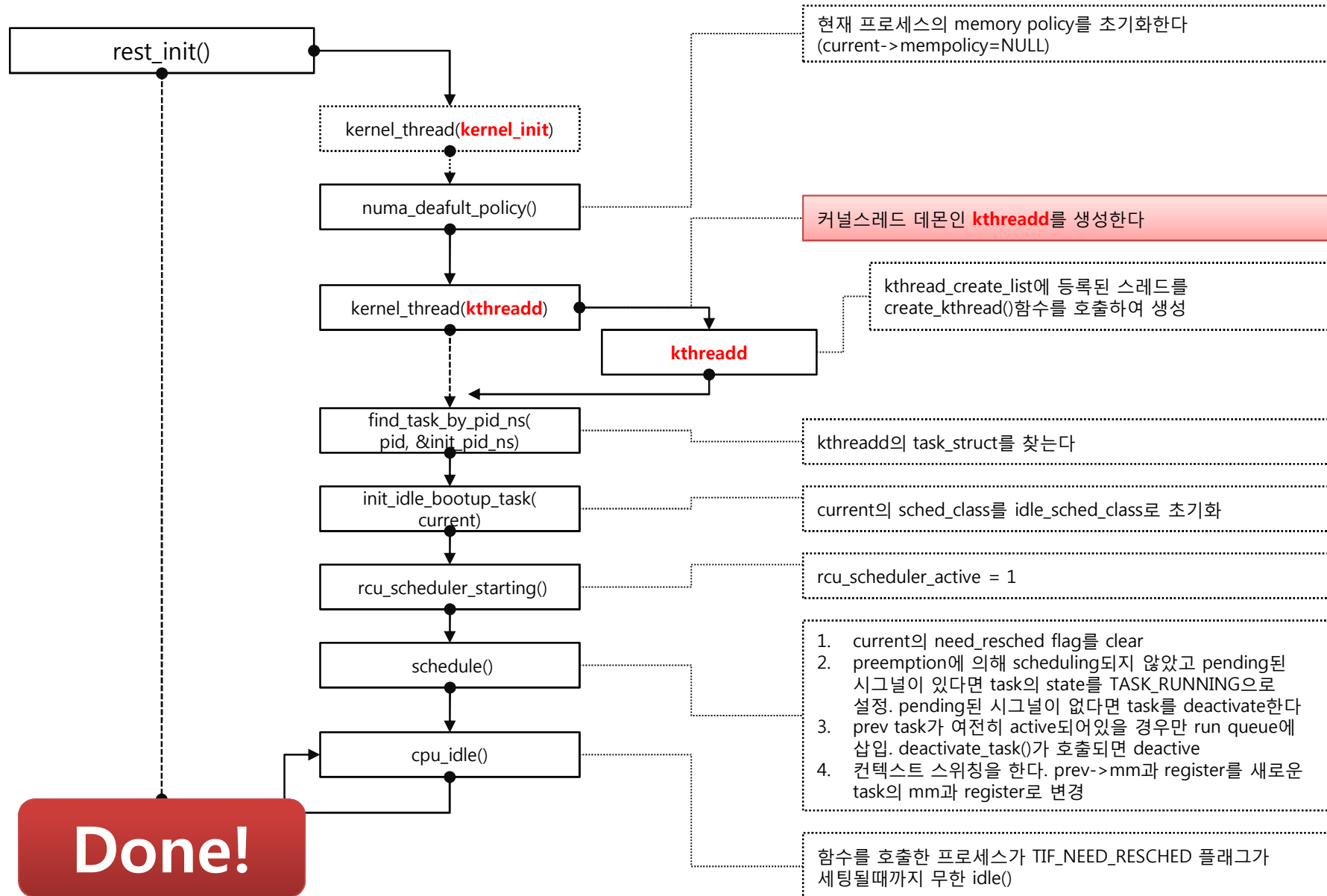
init/main.c



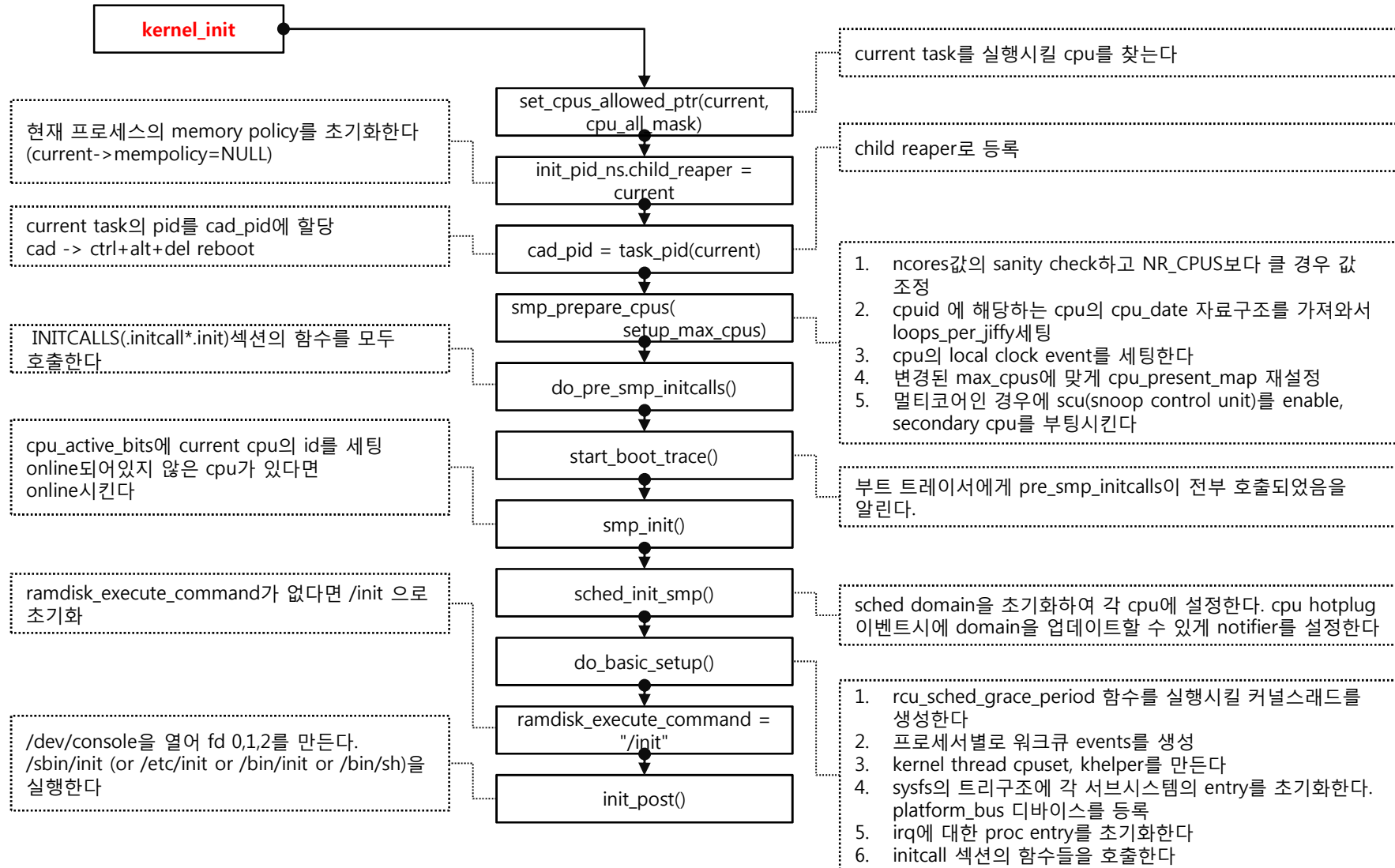
init/main.c



init/main.c



init/main.c



init/main.c

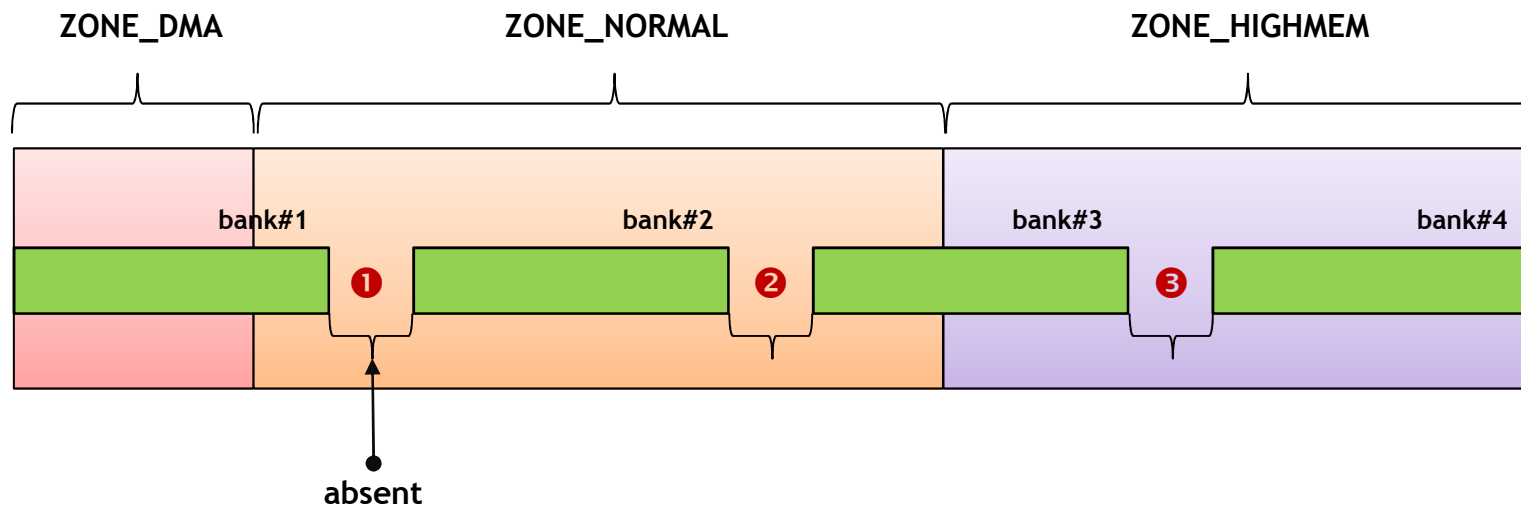
kernel init 함수의 주요 Operation

1. current task를 실행시킬 cpu를 찾고 current를 child reaper로 등록
2. current task의 pid를 cad_pid에 할당. cad -> ctrl+alt+del reboot
3. ncores값의 sanity check하고 NR_CPUS보다 클 경우 값 조정. cpuid 에 해당하는 cpu의 cpu_date 자료구조를 가져와서 loops_per_jiffy세팅. cpu의 local clock event를 세팅한다. 변경된 max_cpus에 맞게 cpu_present_map 재설정. 멀티코어인 경우에 scu(snoop control unit)를 enable, secondary cpu를 부팅시킨다
4. INITCALLS(.initcall*.init)섹션의 함수를 모두 호출한다
5. 부트 트레이서에게 pre_smp_initcalls이 전부 호출되었음을 알린다.
6. cpu_active_bits에 current cpu의 id를 세팅. online되어있지 않은 cpu가 있다면 online시킨다
7. sched domain을 초기화하여 각 cpu에 설정한다. cpu hotplug 이벤트시에 domain을 업데이트할 수 있게 notifier를 설정한다
8. rcu_sched_grace_period 함수를 실행시킬 커널스래드를 생성한다. 프로세서별로 워크큐 events를 생성. kernel thread cpuset, khelper를 만든다. sysfs의 트리구조에 각 서브시스템의 entry를 초기화한다. platform_bus 디바이스를 등록. irq에 대한 proc entry를 초기화한다. initcall 섹션의 함수들을 호출한다.
9. ramdisk_execute_command 검사
10. /dev/console을 열어 fd 0,1,2를 만든다
11. init을 실행한다

Done!

Zone_size, zhone_size 배열

4개의 Bank로 구성된 Node의 3개의 Zone



zone_size[0] = size of ZONE_DMA
zone_size[1] = size of ZONE_NORMAL
zone_size[2] = size of ZONE_HIGHMEM

zhole_size[0] = ZONE_DMA의 영역중 hole size
zhole_size[1] = ① + ②
zhole_size[2] = ③

Memory Bank, Node, NUMA, UMA, ZONE, Page Frame (1)

1. Memory Bank

- Memory Slot의 구성되어 있음.
- 각 Slot에는 같은 타입의 메모리가 장착된 것을 Memory Bank라 함.

2. Node

- 한 CPU로부터 접근속도가 같은 메모리의 집합을 노드라고 부른다
- 여러 개의 Bank가 하나의 Node를 구성할 수 있음

3. NUMA vs. UMA*

- Linux에서 Memory Bank를 표현하는 구조
- NUMA는 프로세서로부터 메모리까지의 거리에 따라 메모리에 대한 접근 속도가 다른 아키텍처
- NUMA에서는 커널이 노드간 통신을 최소화할 필요가 있는데, 커널 텍스트와 읽기전용 데이터를 노드간 복사하는 것에서부터 커널이 필요로 하는 데이터구조체를 한 노드에 유지시키려는 방법들이 필요.
- UMA는 접근속도가 같은 아키텍처

4. ZONE

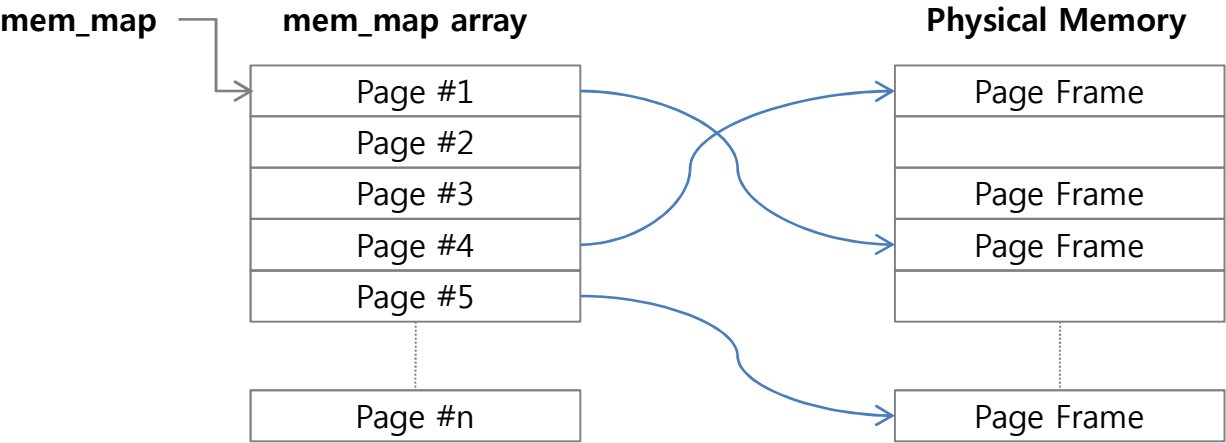
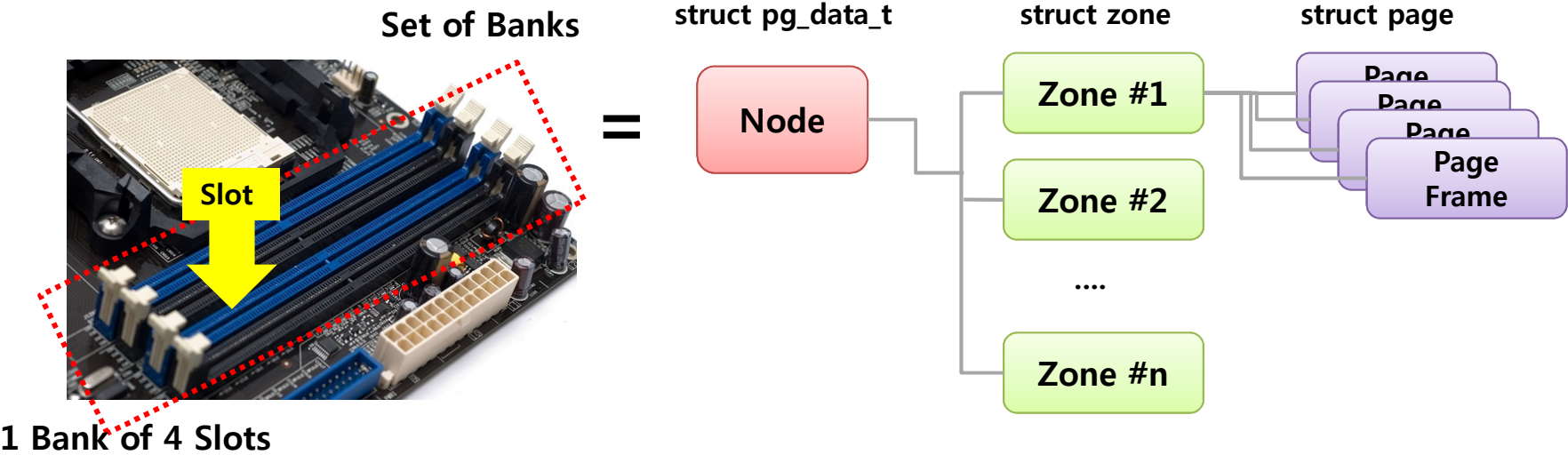
- Node에서 동일한 속성을 가지는 영역
- DMA, NORMAL, HIGHMEM, ZONE_MOVABLE등이 있으나 ARM의 경우는 한 개의 ZONE만 존재

5. Page Frame

- ZONE은 물리 메모리를 관리하며 최소 단위를 Page Frame이라 함.
- Page Frame은 Linux에서 page 구조체로 관리되며, mem_map 전역배열을 통해 접근

*NUMA와 UMA 플랫폼간 코드 일관성을 위해 UMA 플랫폼들은 정적으로 할당된 pg_data_t를 가지고 있다.

Memory Bank, Node, NUMA, UMA, ZONE, Page Frame (2)



Memory Bank, Node, NUMA, UMA, ZONE, Page Frame (3)

Linux에서 Bank가 Node가 아닌 이유

find_bootmap_pfn는 bootmap의 page frame number를 찾는 함수이다. **for_each_nodebank**에서 memory info의 bank들을 순차적으로 돌면서 bank의 node 번호가 입력받은 node 번호와 일치하는지 검사한다. 만약, bank가 node라면 모든 bank는 유일한 node번호를 가져야 할 것이고 Bank의 끝 page frame number가 bss 영역의 끝 page frame number 보다 작은지를 검사하는 **if 문의 continue 문장이 의미가 없어진다**. 그리고 if 문을 수행했을 때 true 라면, **for_each_nodebank** 문이 종료될 것이다.

따라서, 리눅스 커널은 bank와 node를 다르게 보며 한 node는 여러 개의 bank를 가질 수 있다고 유추할 수 있다.

find_bootmap_pfn, arch/arm/mm/init.c

```
for_each_nodebank(i, mi, node) {
    start_pfn = PAGE_ALIGN(__pa(_end)) >> PAGE_SHIFT;
    struct membank *bank = &mi->bank[i];
    unsigned int start, end;

    start = bank_pfn_start(bank);
    end = bank_pfn_end(bank);

    if (end < start_pfn)
        continue;
```

arch/arm/include/asm/setup.h

```
#define for_each_nodebank(iter,mi,no) \
    for (iter = 0; iter < (mi)->nr_banks; iter++) \
        if ((mi)->bank[iter].node == no)
```

Slots

This is the total number of memory upgrade slots (sockets) followed by their configuration. Banks are the way a system addresses memory. A bank must be completely filled with memory modules of the same size and type in order for the system to recognize and address the memory. i.e. :

3 (3 banks of 1) This indicates that there are 3 memory slots. These are divided into 3 banks, and each bank consists of one memory slot. So you can add memory one piece at a time for the system to use.

4 (2 banks of 2) This indicates that there are 4 memory slots. These are divided into 2 banks, and each bank consists of two memory slots. So you must add memory two pieces at a time (they must be the same size and type of memory) in order for the system to benefit from the upgrade.

12 (3 banks of 4) This indicates that there are 12 memory slots. These are divided into 3 banks, and each bank consists of four memory slots. So you must add memory four pieces at a time (and they must be the same size and type of memory) in order for the system to benefit from the upgrade.

Node 구성도

include/linux/mmzone.h

pg_data_t: Memory Node를 표현하는 구조체

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES]; /* Node내 Zone에 대한 Array */
    struct zonelist node_zonelists[MAX_ZONELISTS]; /* Zone에 대한 Fallback List, Zone에 가용 메모리가 없을시 다른 Node의 같은 Type의
                                                    Zone을 찾아 메모리를 할당하기 위함 */
    int nr_zones; /* Node내 Zone의 개수, Node마다 다른 Zone의 개수를 가질 수 있음 */
    struct page *node_mem_map; /* Page Array에 대한 포인터이며 해당 Node에 속한 Page 의 시작 주소임 */
    struct bootmem_data *bdata; /* Memory Allocator가 활성화되지 않은 Boot시에 Kernel이 메모리를 사용할 필요가
                                있음. 이 경우 임시적으로 메모리관리를 하기위한 구조체로 Boot Memory Allocator
                                가 사용함 */
    unsigned long node_start_pfn; /* Logical Page Frame 번호로, 시스템내의 모든 Node의 페이지들은 연속적인 번호가
                                부여되며, Globally Unique한 번호임. */
    unsigned long node_present_pages; /* total physical page 수 */
    unsigned long node_spanned_pages; /* 해당 Node의 모든 page 수, Node내 Hole이 존재할 경우 Physical한 Page가 존재하
                                지 않을 수 도 있음 */
    int node_id; /* Global Node Identifier */
    wait_queue_head_t kswapd_wait; /* swap daemon을 위한 wait queue로 zone에 있는 frame을 swap하기위해 필요 */
    struct task_struct *kswapd; /* swap daemon에 대한 task struct 포인터 */
    int kswapd_max_order; /* swapping subsystem에 사용되며, free 될 영역의 크기를 정할때 사용됨 */
} pg_data_t;
```

struct pglist_data *pgdat_next 필드는 2.6.31에 존재하지 않음.

이전 version에서는 시스템 내 Node들을 Singly Linked List로 연결하기 위해 사용됨.

Node 구성도

include/linux/mmzone.h

```
struct zone {
    unsigned long    pages_min, pages_low, pages_high; /* swap을 위한 기준, 아래 참고 */
    unsigned long    lowmem_reserve[MAX_NR_ZONES]; /* critical allocation을 위해 예약, 어떠한 조건에서도 할당되어야 할경우 사용 */
    struct free_area  free_area[MAX_ORDER]; /* Buddy System의 근간 */

    /* Fields commonly accessed by the page reclaim scanner */
    spinlock_t    lru_lock;
    struct {
        struct list_head list;
        unsigned long nr_scan;
    } lru[NR_LRU_LISTS];

    struct zone_reclaim_stat reclaim_stat;

    unsigned long    pages_scanned; /* since last reclaim */
    unsigned long    flags; /* zone flags */

    atomic_long_t    vm_stat[NR_VM_ZONE_STAT_ITEMS]; /* Zone Statistics */
    int prev_priority; /* Page reclaim에 의해 page frame이 free될 때
                        zone이 scan되었을 때 Priority */

    unsigned int inactive_ratio;

    wait_queue_head_t * wait_table; /* page가 available할 때까지 기다리고 있는 process들을 위한 */
    unsigned long    wait_table_hash_nr_entries; /* wait queue를 구현하기 위한 변수 들 */
    unsigned long    wait_table_bits;

    struct pglist_data *zone_pgdat; /* zone이 속한 node에 대한 포인터 */
    unsigned long    zone_start_pfn; /* zone의 첫 page frame number */

    unsigned long    spanned_pages; /* total size, including holes */
    unsigned long    present_pages; /* amount of memory (excluding holes) */
    const char    *name;
} ____cacheline_internodealigned_in_smp;
```

if (free pages > pages_high)
OK.
else if (free pages < page_low)
swapping
else if (free_pages < pages_min)
page recalim

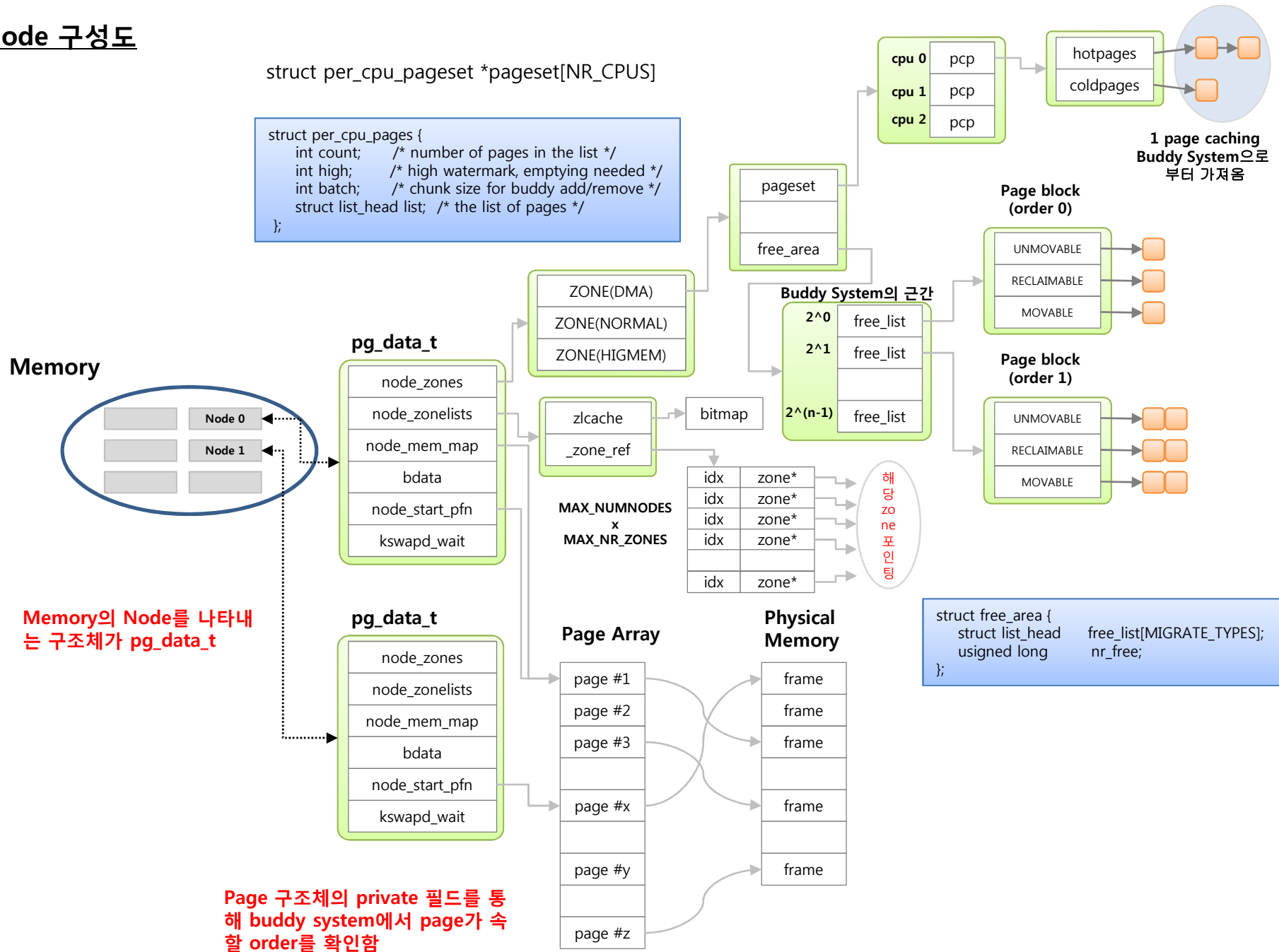
Zone의 flag

```
typedef enum {
    ZONE_ALL_UNRECLAIMABLE, /* all pages pinned */
    ZONE_RECLAIM_LOCKED, /* prevents concurrent reclaim */
    ZONE_OOM_LOCKED, /* zone is in OOM killer zonelist */
} zone_flags_t;
```

의미

if (flags is Nothing)
NORMAL
else if (flags == ZONE_ALL_UNRECLAIMABLE)
all pages are pinned, so recalim 불가
else if (flags == RECLAIM_LOCKED)
다른 CPU가 reclaim을 하지 못하도록 함, 현재 recalim 중
else if (flags == ZONE_OOM_LOCKED)
process가 과다한 memory를 사용하고 있고 operation을 완료할 수
없는 경우 Out of Memory를 방지하기 위함

Node 구성도



Node 구성도

Buddy System Information & Page Type Information

```
File Edit View Terminal Help
rsyoung@b.russell:~$ cat /proc/buddyinfo
Node 0, zone DMA 4 3 1 2 2 2 0 0 1 1 3
Node 0, zone DMA32 4071 2280 980 164 23 11 3 2 0 0 0
Node 0, zone Normal 636 45 41 2 2 0 0 0 0 0 0
rsyoung@b.russell:~$ cat /proc/pagetypeinfo
Page block order: 9
Pages per block: 512

Free pages count per migrate type at order 0 1 2 3 4 5 6 7 8 9 10
Node 0, zone DMA, type Unmovable 4 3 1 2 2 2 0 0 1 0 0
Node 0, zone DMA, type Reclaimable 0 0 0 0 0 0 0 0 0 0 0
Node 0, zone DMA, type Movable 0 0 0 0 0 0 0 0 0 0 3
Node 0, zone DMA, type Reserve 0 0 0 0 0 0 0 0 0 1 0
Node 0, zone DMA, type Isolate 0 0 0 0 0 0 0 0 0 0 0
Node 0, zone DMA32, type Unmovable 82 87 23 3 2 1 0 0 0 0 0
Node 0, zone DMA32, type Reclaimable 596 64 136 1 0 0 0 0 0 0 0
Node 0, zone DMA32, type Movable 3393 2129 821 159 20 10 2 1 0 0 0
Node 0, zone DMA32, type Reserve 0 0 0 0 1 0 1 1 0 0 0
Node 0, zone DMA32, type Isolate 0 0 0 0 0 0 0 0 0 0 0
Node 0, zone Normal, type Unmovable 454 20 2 0 0 0 0 0 0 0 0
Node 0, zone Normal, type Reclaimable 43 9 0 0 0 0 0 0 0 0 0
Node 0, zone Normal, type Movable 162 15 39 1 0 0 0 0 0 0 0
Node 0, zone Normal, type Reserve 4 1 1 1 2 0 0 0 0 0 0
Node 0, zone Normal, type Isolate 0 0 0 0 0 0 0 0 0 0 0

Number of blocks type Unmovable Reclaimable Movable Reserve Isolate
Node 0, zone DMA 1 0 6 1 0
Node 0, zone DMA32 348 27 1151 2 0
Node 0, zone Normal 102 6 403 1 0
rsyoung@b.russell:~$
```

첫 번째 0 bit를 찾는 함수

arch/arm/lib/findbit.S

```
ENTRY(_find_first_zero_bit_le)
    teq r1, #0
    beq 3f
    mov r2, #0
1:   ldrb r3, [r0, r2, lsr #3]
    eors r3, r3, #0xff @ invert bits
    bne .L_found @ any now set - found zero bit
    add r2, r2, #8 @ next bit pointer
2:   cmp r2, r1 @ any more?
    blo 1b
3:   mov r0, r1 @ no free bits
    mov pc, lr
ENDPROC(_find_first_zero_bit_le)

.L_found:
#if __LINUX_ARM_ARCH__ >= 5
    rsb r1, r3, #0 @ r1 = 0 - r3
    and r3, r3, r1 @ r3 = r1 & r3 이 연산이 후, r3의 값은
                    @ bit 값이 0인 첫번째 bit만 1로 셋트됨
                    @ 나머지는 모두 0 예를 들면
                    @ r3 = 00000000 00000000 00000000 00000010

    clz r3, r3 @ 첫 번째 1을 만날때까지 0을 카운트
    rsb r3, r3, #31 @ 31 - r3, 왜 31이냐?
                    @ bitmap의 해당 byte에는 적어도 1개의
                    @ 1이 존재할 것이기 때문에 0 개수의 max
                    @ 값은 31. 처음으로 0인 bit에 대한
                    @ offset을 구할 수 있음
    add r0, r2, r3 @ base 값 (r2)에 offset을 더해 return
#else
    ....
#endif
    mov pc, lr
```

_find_first_zero_bit_le 는 _find_next_zero_bit_le와 매우 유사

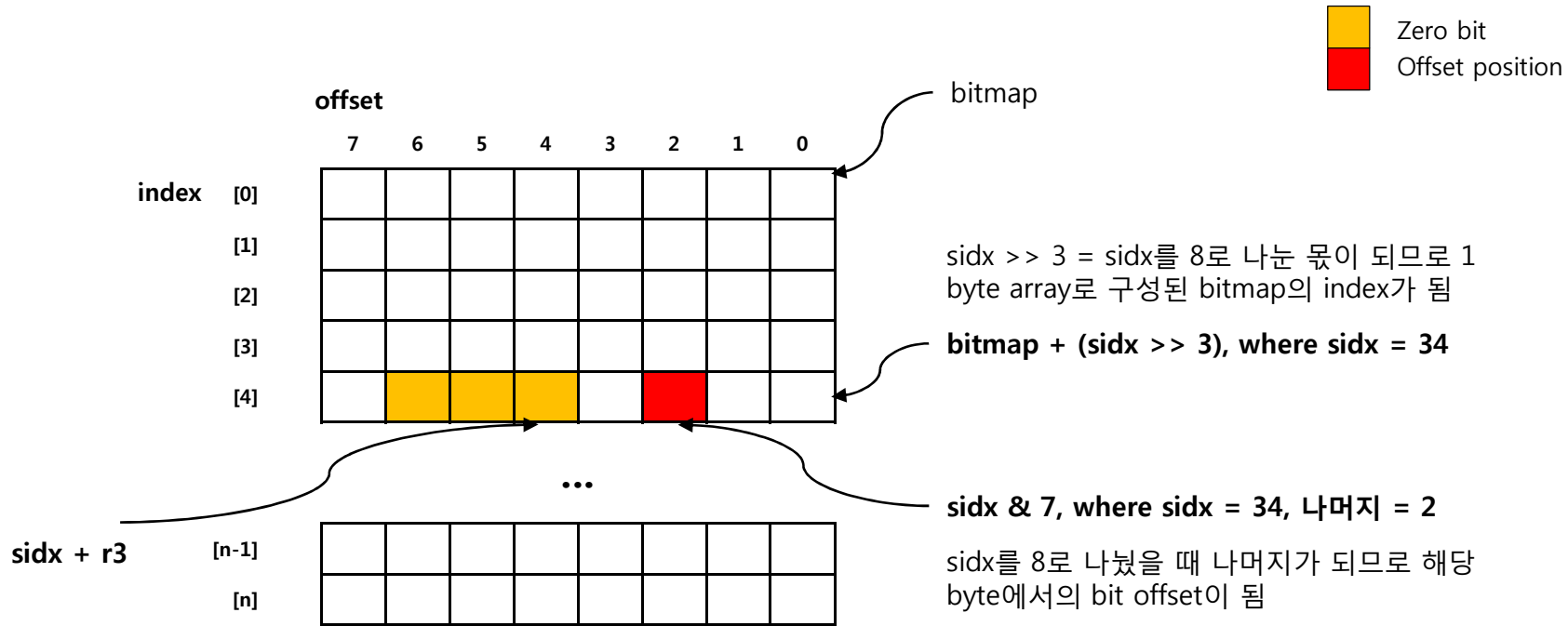
```
ENTRY(_find_next_zero_bit_le)
    teq r1, #0 @ maxbit (size)가 0이라면
    beq 3b @ backward 3로 점프
    ands ip, r2, #7 @ r2 즉 sidx가 8의 배수인지를 검사
                    @ 8의 배수라면 _find_first_zero_bit_le 로
                    @ bitmap이 8bit씩 구성되어 있기 때문에,
                    @ 8의 배수일 경우는 한 바이트의 첫 번째
                    @ 비트부터 검사할 수 있지만, 8의 배수가 아닐
                    @ 경우는 8로 나누었을 때 나머지 값에 대한
                    @ 처리를 해줘야 합니다.
    beq 1b @ 8의 배수일 경우 _find_first_zero_bit_le

    ldrb r3, [r0, r2, lsr #3] @ 액세스할 메모리 주소를 계산함
                    @ r2, lsr #3에 의해 기준점 r0로 부터
                    @ byte 단위 offset을 구함.

    eor r3, r3, #0xff @ r3과 0xff를 EOR해서 결과값을 r3에 저장
                    @ r3이 0이 아라면 원래 r3의 값에는 0 bit가
                    @ 존재한다는 의미

    movs r3, r3, lsr ip @ r2를 8로 나누었을 때 나머지 값 만큼
                    @ 오른쪽으로 쉬프트 시킴.
                    @ 나머지 값에 해당되는 offset을
                    @ 건너뛰어야 하기 때문
                    @ 예를 들어, ip가 6이라면, 6번 이상의
                    @ bit들 중 0인 값을 찾아야 함.
    bne .L_found @ 여기까지 왔는데, r3 값이 여전히 0이 아니라면,
                    @ r3에는 0을 나타내는 비트가 존재한다는
                    @ 의미이고 페이지를 할당할 수 있음.
                    @ 할당할 수 있는 페이지가 있으므로,
                    @ .L_found로 점프
    orr r2, r2, #7 @ if zero, then no bits here
    add r2, r2, #1 @ align bit pointer
    b 2b @ loop for next bit
ENDPROC(_find_next_zero_bit_le)
```


Bootmem bitmap에서 sidx를 기준으로 최초 0 bit를 찾는 로직



Step 1	Index [4] = r3 =	1	0	0	0	1	1	1	1	
Step 2	0xff =	1	1	1	1	1	1	1	1	
Step 3	r3 = r3 XOR 0xff =	0	1	1	1	0	0	0	0	
Step 4	r3 = r3 >> (sidx & 7) =	0	0	0	1	1	1	0	0	
Step 5	r1 = 0 - r3 =	1	1	1	0	0	1	0	0	
Step 6	r3 = r1 and r3 =	0	0	0	0	0	1	0	0	
Step 7	clz r3, r3 = 29, (00000000 00000000 00000000 00000100)									
Step 8	r3 = 31 - r3 = 2									
최초 가용bit의 위치 = sidx + r3										

[step 3] R3 XOR 0xff가 00이 아니라면, r3가 0 bit를 포함

[step 4] 2 bit shift, r3가 00이 아니라면 r3에 0 bit가 존재

[step 5] 최하위 bit부터 시작하여 최초 bit 1의 위치만 1로 셋팅 됨 이 의미는 step 4에서 첫번째 1이 되는 bit를 의미하고, 그 1은 step 1에서 첫번째 가용한 bit 0를 나타냄.

[Step 7] leading zero를 counting 함 r3 = 29

[Step 8] 최소 1개의 1bit는 1이 있을 것이므로 0의 최대 개수는 31에서 r3를 빼면, sidx + r3는 index[4]에서 최초 가용 bit를 가리키게 됨

Memory Hotplug 정리

1. Purpose

- A. On Demand에 대응하여 메모리의 양을 변경, 이 경우는 주는 highly virtualized environments에서 요구됨
- B. DIMMs나 NUMA-nodes를 물리적으로 Install하거나 제거하고자 할 때, 이 경우는 메모리 파워관리를 지원하고자 하는 하드웨어에서 요구됨.

2. 2 Phases

• 물리적 메모리 핫플러그 단계

- 하드웨어나 펌웨어와 커뮤니케이션을 하여 핫플러그된 메모리를 위한 환경을 만들거나 제거한다.
- 목적 (B)를 위한 것이지만, highly virtualized environments간 커뮤니케이션을 위한 좋은 단계다.
- 메모리가 핫플러그될 때 커널은 새로운 메모리를 인지하고 새로운 메모리 관리 테이블 (Memory Management Table)을 만들고 새로운 메모리에 대한 오퍼레이션을 위해 시스파일시스템 파일들을 만든다.
- 펌웨어가 새로운 메모리가 추가되었다고 OS에 Noti를 할 수 있다면, 이 단계는 자동적으로 트리거된다.
- ACPI는 이 이벤트를 Noti할 수 있다. 만일 펌웨어에 그런 기능이 없다면, 시스템 관리자에 의한 "probe" 오퍼레이션이 대신 사용된다.

• 논리적 메모리 핫플러그 단계

- 논리적 메모리 핫플러그 단계에서는 메모리 상태를 "가용"또는 "불가용" 상태로 변경한다.
- 이 단계에서 사용자의 메모리 양이 변하게된다.
- 커널은 이 단계에 있는 모든 메모리를 메모리 범위가 가용할 때 프리페이지로 만든다.
- 이 단계를 online/ offline라고 함.
- 시스템 관리자가 sysfs 파일에 쓰기 오퍼레이션을 하면 논리적 메모리 핫플러그 단계가 트리거된다.
- hot-add의 경우는 물리적 핫플러그 단계 후에 수행되어야 한다. (하지만, 메모리 핫플러그를 위해 udev의 핫플러그 스크립트에 쓴다면, 이 단계는 구분없이 수행될 수 있다.)

Memory Hotplug 정리

3. Logical Memory Remove

- 메모리 오프라인은 전체 메모리 섹션을 비가용상태로 만들어야 하기 때문에 만일 섹션이 메모리를 해제시킬 수 영역을 포함하고 있다면 메모리 오프라인은 실패함
- 일반적으로 메모리 오프라인에는 다음과 같은 2가지 기술이 사용된다.
 - (1) 섹션내에 있는 모든 메모리를 회수하여 해제시킨다.
 - (2) 섹션에 있는 모든 페이지를 이주시킨다.
- 현재 리눅스의 메모리 오프라인은 페이지를 이주시켜 섹션에 있는 모든 페이지를 해제하는 방법인 방법 (2)를 사용한다.
- 현재 리눅스에서 이주 가능한 페이지들은 모두 anonymous 페이지와 페이지 캐시들이다.
- 페이지 이주에 의한 섹션을 오프라인 시키기 위해서 섹션이 모두 이주 가능한 페이지들만을 포함하고 있다는 것을 커널이 보장해야 한다.
- 섹션을 이주 가능한 페이지로만 구성되도록 하는 부트 옵션이 현재 지원되고 있다. "kernelcore="나 "movablecore=" 부트 옵션을 정의하여 이동 가능한 페이지들을 위한 zone인 ZONE_MOVABLE을 생성.
- ZON_MOVABLE에 있는 섹션은 쉽게 오프라인 될 수 있는 것으로 간주되지만 가끔 busy 상태에서는 -EBUSY가 리턴된다. 메모리 섹션이 -EBUSY 때문에 오프라인이될 수 없을 지라도, 몇 번 시도하면 메모리 섹션을 오프라인 시킬 수도 있다. (예를 들면, 커널내부 호출에 의해 참조되는 페이지는 조만간 릴리즈 될 수 있다.)

4. Memory Hotplug Event Notifier

- MEMORY_GOING_ONLINE
 - 서브시스템이 메모리를 핸들링 할 수 있도록 하기 위해 새로운 메모리가 가용상태가 되기 전에 발생됨. 페이지 할당자는 아직 새로운 메모리로부터 할당하지 못함.
- MEMORY_CANCEL_ONLINE
 - MEMORY_GOING_ONLINE이 실패할 때 발생됨.
- MEMORY_ONLINE
 - 메모리가 성공적으로 온라인 상태가 되었을 때 발생됨. callback은 새로운 메모리로부터 페이지를 할당할 수 있음.
- MEMORY_GOING_OFFLINE
 - 메모리를 오프라인하기 위해 발생됨. 메모리 할당은 더 이상 불가하나 오프라인 하고자 하는 메모리의 일부는 여전히 사용 중일 수 있음. callback은 지정된 메모리 섹션에서 서브시스템에 알려진 메모리를 해제하기 위해 사용될 수 있다.
- MEMORY_CANCEL_OFFLINE
 - MEMORY_GOING_OFFLINE이 실패할 경우 발생됨. 오프라인을 시도했던 섹션에 대한 메모리가 다시 가용상태가 됨. MEMORY_OFFLINE 메모리 오프라인이 완료된 후에 발생함.

Thread Model

■ 1:1 (Kernel-level threading)

- User가 생성한 Thread가 Kernel에서 스케줄될 Entity로 1:1 매핑
- Thread 구현을 심플하게 함.
- NPTH와 LinuxThread가 1:1 Model 임.
- Solaris, NetBSD and FreeBSD에서 1:1 Model 사용

■ N:1 (User-level threading)

- 모든 Application-level Thread들은 하나의 Kernel-level Scheduling Entity로 매핑
- Kernel은 Application Thread에 대한 정보를 전혀가지고 있지 않음.
- Context Switching을 매우 빠르게 할 수 있으나 Multi-threaded processor나 Multi-Processor Computer와 같은 하드웨어적 Benefit을 전혀 얻을 수 없음. 즉, 동시에 한 개 이상의 Thread가 Schedule될 수 없음. GNU Portable Thread에서 사용됨.

■ N:M (Hybrid threading)

- N개의 Application Thread가 M개의 Schedule가능한 Thread로 매핑됨.
- Thread Library가 User Thread에 대한 Scheduling을 책임지고 Context Switching이 매우 빠른 장점이 있음 (System Call을 하지 않기 때문)
- 구현이 매우 어려운 단점이 있고 Priority가 바뀔수 있는 가능성이 있으며, User Space의 Scheduling과 Kernel Space의 Scheduling이 Coordinate되어야 함.

LinuxThread vs. NPTL vs. NGTL

■ LinuxThread

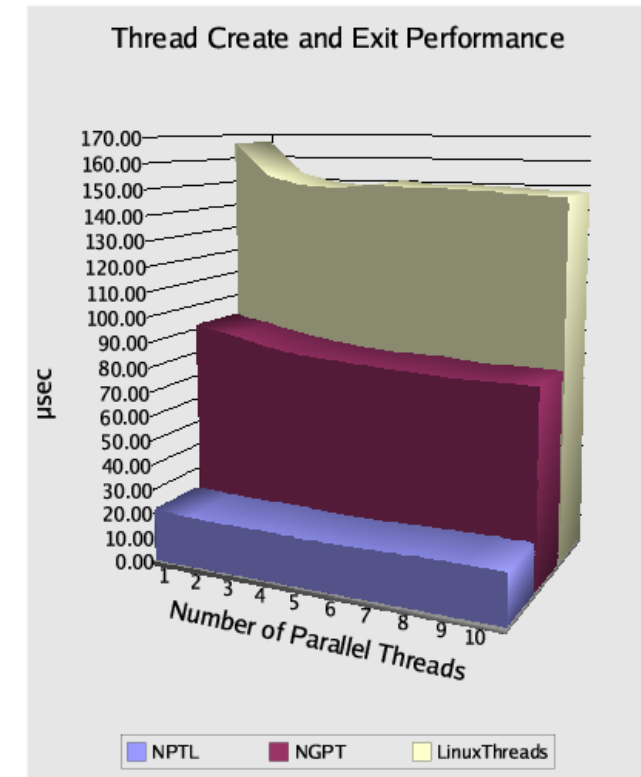
- Clone system call을 사용하여 새로운 process를 생성. 이 호출은 호출하는 프로세스의 복사본을 생성해 호출자의 주소 영역에서 복사 공유가 가능하게 만들었다. LinuxThreads 프로젝트는 이런 시스템 호출을 사용해 사용자 영역에서 스레드 지원을 완벽하게 흉내 내었다. POSIX Threads를 부분적으로 구현한 것
- 새로운 process는 process identifier를 갖게 되어 Signal Handling 문제가 발생함.
- Thread들간 통신은 SIGUSR1과 SIGUSR2를 사용함으로써 프로그램에서 이 signal들을 사용할 수 없었음.
- 2.6 이전에는 진정한 thread를 지원하지 못했고, clone system call을 이용하여 지원. 따라서, 이 시기의 pthread는 모두 user space에서 이루어진 것임. signal handling, scheduling, inter-process synchronization 문제를 내포함.
- 이러한 문제점을 해결하기 위해 두 프로젝트가 시작됨
 - NGPT: Next Generation POSIX Threads (IBM 주도)
 - NPTL: Native POSIX Thread Library (RedHat 주도)
- NPTL에 의해 대체됨.

■ NPTL (Native POSIX Thread Library)

- LinuxThreads와 같이 NPTL은 1대1 모델을 구현한다
- NPTL이 POSIX 규약을 따르므로 NPTL은 프로세스 단위로 시그널을 처리한다. getpid()는 모든 스레드에서 똑같은 프로세스 ID를 반환한다. 예를 들어 시그널 SIGSTOP을 보내면 전체 프로세스가 멈춘다. LinuxThreads에서는 이 시그널을 받은 스레드만 멈춘다. 이는 NPTL 기반 응용에서 GDB와 같은 디버깅 지원을 강화한다.

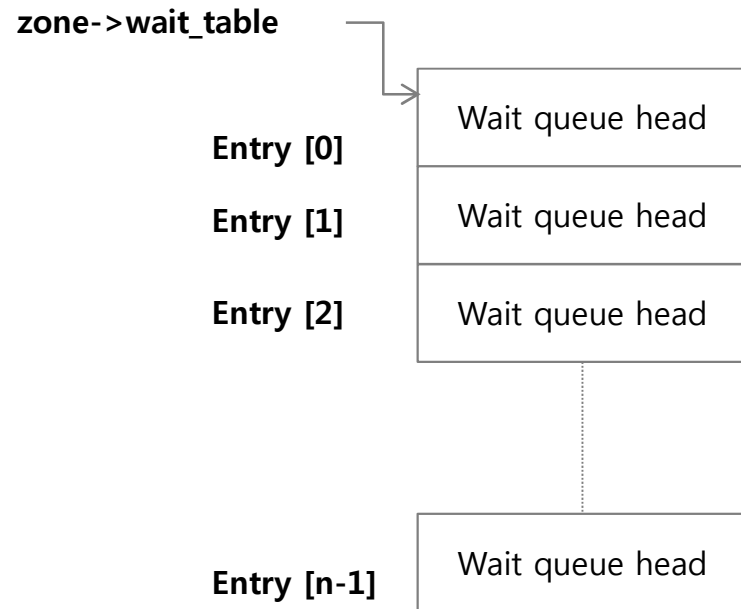
■ NGPT (Next-Generation POSIX Threads)

- IBM 주도로 LinuxThread를 대체하기 위해 시작됨.
- N:M Model



people.redhat.com/drepper/nptl-design.pdf

Zone의 Wait Table 구성



Each entry는 wait queue head

```
struct __wait_queue_head {  
    spinlock_t lock;  
    struct list_head task_list;  
};
```

`zone->wait_table_hash_nr_entries = n-1`

`zone->wait_table_bits = log2(n)`

Buddy System

- MAX_ORDER는 총 몇 개의 order가 있는지 나타내는 것임.
- 즉, 2^n 에서 $n = \text{order}$
- Order는 0 ~ MAX_ORDER까지 임
- The indices of the individual elements of the free_area[] array are also interpreted as order parameters and specify how many pages are present in the contiguous areas on a shared list. The zeroth array element lists sections with one page ($2^0 = 1$), the first lists page pairs ($2^1 = 2$), the third manages sets of 4 pages, and so on.
- Memory management based on the buddy system is concentrated on a single memory zone of a node, for instance, the DMA or high-memory zone. However, the buddy systems of all zones and nodes are linked via the allocation fallback list. Figure 3-23 illustrates this relationship.

```
<mmzone.h>
struct zone {
...
    /*
     * free areas of different sizes
     */
    struct free_area free_area[MAX_ORDER]
...
};
```

```
<mmzone.h>
struct free_area {
    struct list_head free_list[MIGRATE_TYPES]
    unsigned long nr_free;
};
```

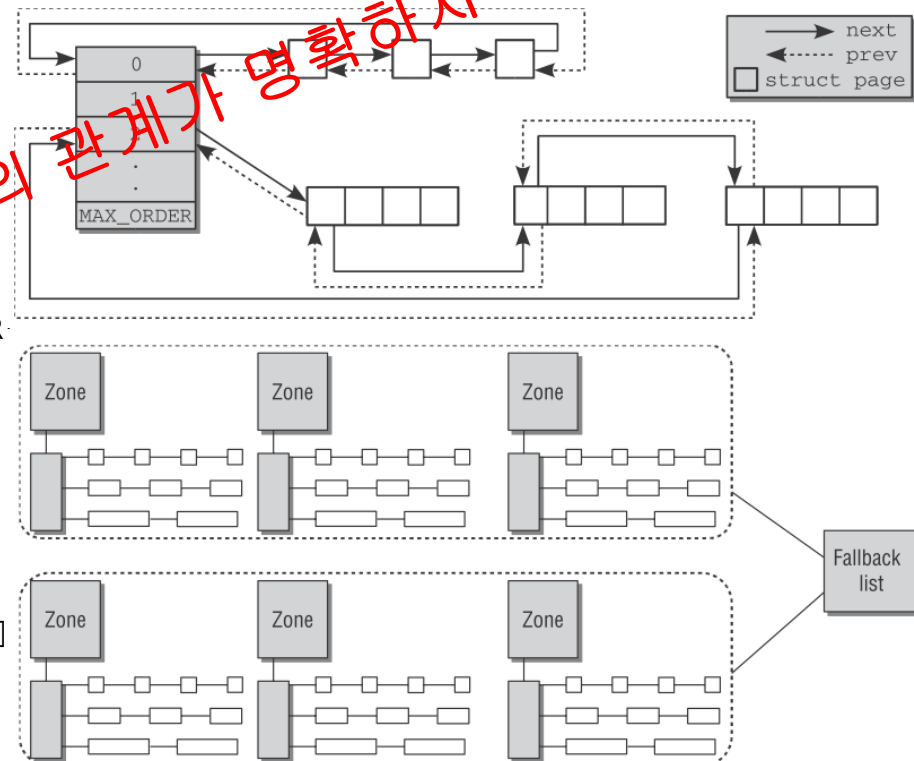


Figure 3-23: Relationship between buddy system and memory zones/nodes.

Buddy System

- Linux Kernel의 Memory Fragmentation 에 대한 정책은 File System에서 사용하는 defragmentation이 아닌, Anti-Fragmentation임
- Three types of pages
 - Non-movable Pages – Memory에 고정됨
 - Reclaimable Pages – 이동될 수는 없지만 제거될 수 는 있음. Data mapped from files, kswapd daemon에 의해 주기적으로 제거됨.
 - Movable Pages – 이동될 수 있음. Userspace Application으로 page table에 의해 mapping됨
- Anti-Fragmentation의 핵심은 Page의 Mobility를 기준으로 같은 것 끼리 Grouping하는 것임.

같은 mobility 속성을 가지는 page들을 block단위로 묶어 관리하는 것으로 unmovable 속성을 가지는 2개의 연속된 page가 필요하면 order = 1인 unmovable bit가 on된 page block을 찾아 할당하게 되므로 Page Fragmentation을 ****방지**** 할 수 있음. 여기서 ****방지****의 의미는 Linux Kernel이 Fragmentation 해결방법으로 Anti-Fragmentation을 사용하기 때문이며 파일 시스템에서 Defragmentation 방법과는 차이가 있음.

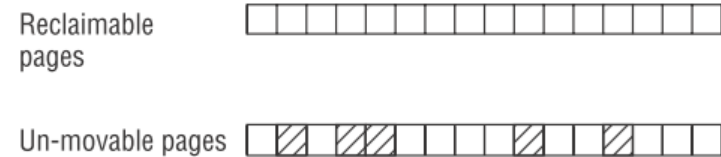
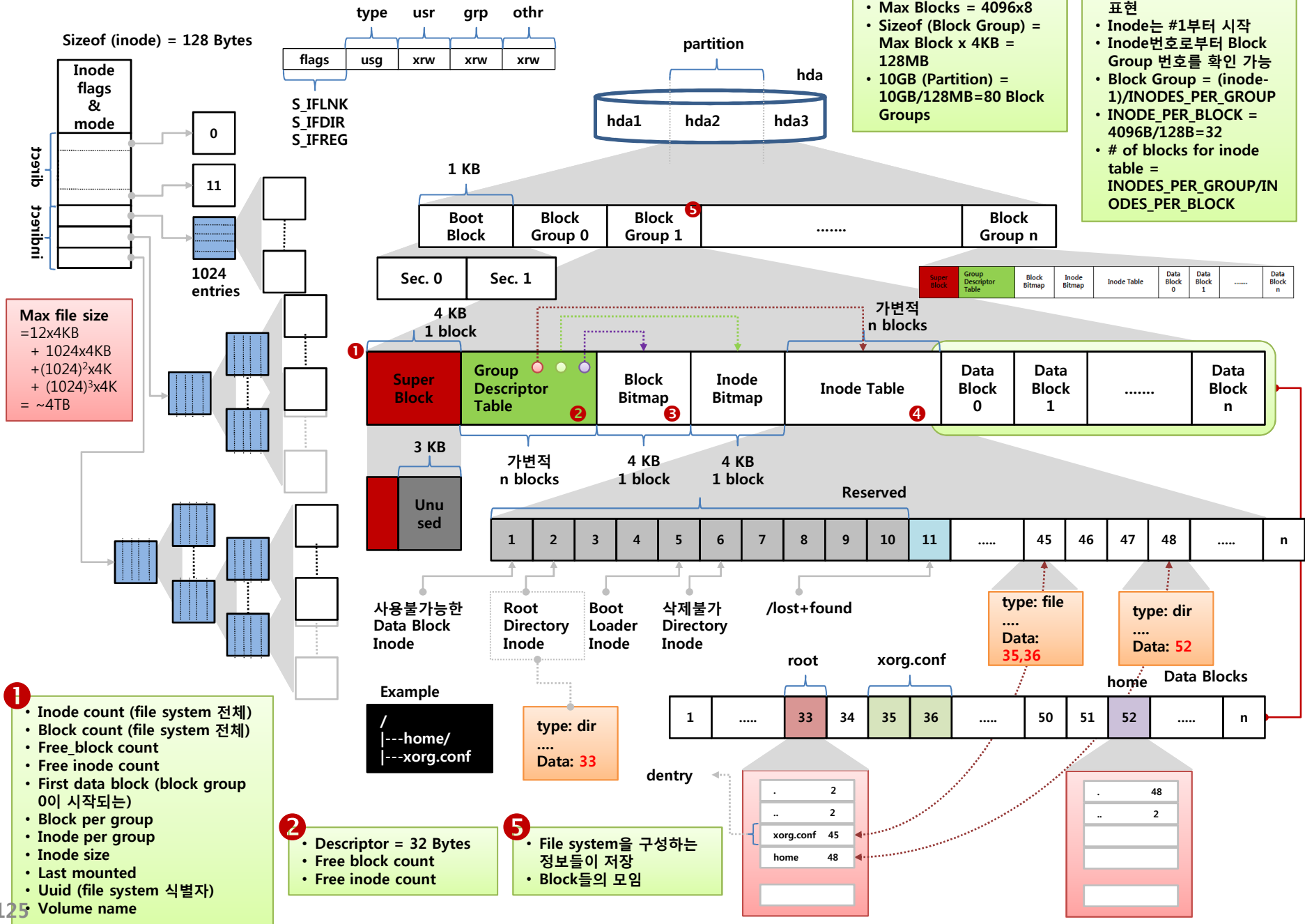


Figure 3-26: Memory fragmentation is reduced by grouping pages together depending on their mobility.

Todo: 정리

File System (Ext2)



Cache

Cache Line Accessing

Address를 세 부분으로 나눠 Cache Line에 접근



1. Index

- 주소값의 중간값을 가져와 Cache Line의 Index로 사용
- 512개의 Cache Line이라면 9 bit 사용 ($2^9=512$)

2. Tag

- Cache Line에 원하는 값이 있는지 검사하기 위해 사용
- 서로 다른 주소값이 동일한 Cache Line을 가르킬 수 있기 때문

3. Offset

- Cache Line내에서 원하는 데이터를 가르키기 위해 사용
- 64 Byte Cache Line일 경우 Offset은 6 bit 사용

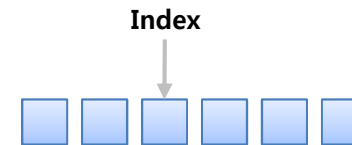
Index를 가운데에 위치 시킨 이유는 Hash를 이용한 Cache Line을 매핑할 시 충돌을 적게 하기 위함

Cache Allocation

Conflict에 의한 Cache Miss를 해결하기 위해 한 Index에 여러 개의 Cache 공간 할당

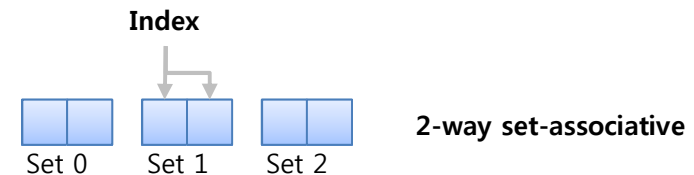
1. Directly-Mapped Cache

- 하나의 Index가 하나의 Cache Line과 Mapping
- 두 데이터가 같은 Cache Index를 가지고 번갈아가며 접근될 경우 Cache Miss가 발생



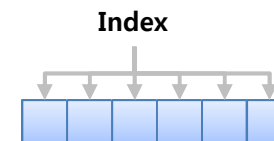
2. Set-Associative Cache

- 하나의 Index가 N개의 Cache Line에 들어갈 수 있음



3. Fully-Associated Cache

- Cache Line어디에도 Mapping될 수 있음



Cache

Cache Replacement Policy

1. LRU
 - Least Recently Used
2. LFU
 - Least Frequently Used
3. FIFO
 - First In First Out

Multi Cache Level

1. L1 Cache
 - Hit Latency를 줄이기 위해 중점
 - 32KB, 64KB 수준
2. L2 & L3 Cache
 - Miss Rate & Missing Penalty를 줄이기 위해 중점
 - 8MB, 12MB

Cache Writing Policy

1. CPOLICY UNCACHED
 - Cache를 사용하지 않음
2. CPOLICY BUFFERED
 - Cache를 사용하지 않고 Cache와 Memory사이의 Write Buffer 사용
3. CPOLICY WRITETHROUG
 - Cache쓰기가 발생할 때 메모리 및 아래 계층에 바로 반영
 - USB 메모리처럼 탈부착이 빈번한 저장 장치의 경우 사용
4. CPOLICY WRITEBACK
 - 변경된 부분을 일단 Cache에 보관하고 Cache Replacement 대상이 될 때 메모리에 갱신
5. CPOLICY WRITEALLOC
 - Cache Miss시 Cache Line 할당 방법
 - 두 가지가 있음
 - 1) read-allocate 방식: 데이터를 Memory에서 읽었을 때 Cache Line 할당
 - 2) write-allocate 방식: Memory에 데이터를 쓸 때 캐시 라인을 할당

Cache

Cache Miss Types

1. Cold Miss

- Compulsory Miss라고도 함
- 데이터를 최초로 읽을 때 발생하는 Cache Miss

2. Conflict Miss

- Cache의 연관도가 부족해 발생하는 Cache

3. Capacity Miss

- Cache의 용량이 부족해 발생하는 Cache Miss

4. Coherence Miss

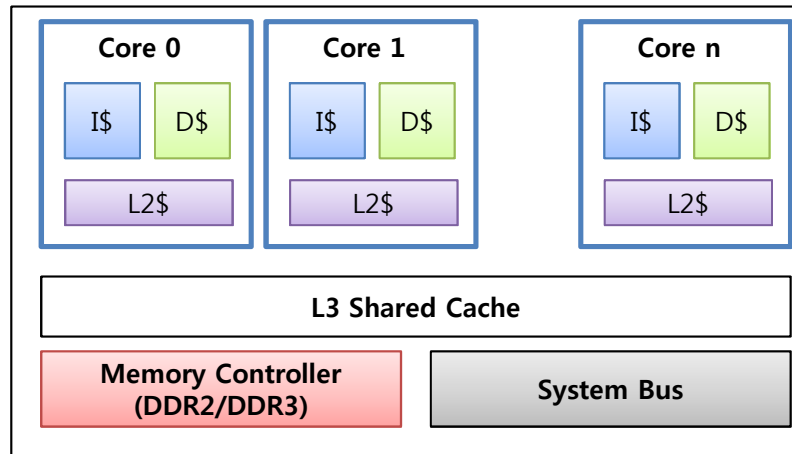
- 다른 프로세서에 의해 Cache Line이 Invalidation되어 발생하는 Cache Miss

Cache Coherence는 어떤 주소의 내용을 읽을 때 항상 최신 내용을 읽을 수 있게 한다. MSI Snooping Protocol은 Cache Coherence의 가장 고전적인 해결 방법으로 Cache Line마다 Coherence 상태인 M (Modified), S(Shared), I (Invalid) 상태를 가지게 한다. 그리고 Cache에 접근할 때마다 모든 캐시에 Bus를 통해 신호를 보내는 Snooping 작업을 한다.

Cache Line이 Processor A에 의해 Invalid된 상태에서 동일한 Cache Line을 Processor B가 접근할 때 Cache Miss발생, 특히 이 경우를 Coherence Miss라고 함

Multi Core에서 Cache

AMD (Phenom)/Intel(Nehalem) Cache 구성도



Cache Coherence 문제가 발생

MSI Snooping Protocol

1. Invalid

- 어떤 Cache Line의 상태가 유효하지 않다. 읽거나 쓰려면 반드시 값을 요청해야 함

2. Shared

- Cache Line이 한 곳 이상에서 공유 중이다. 어떤 Processor가 쓰기를 하려면 반드시 신호를 보내 자신의 Cache Line을 M 상태로 바꾸면서 다른 Cache 사본은 I 상태로 바뀌어야 함

3. Modified

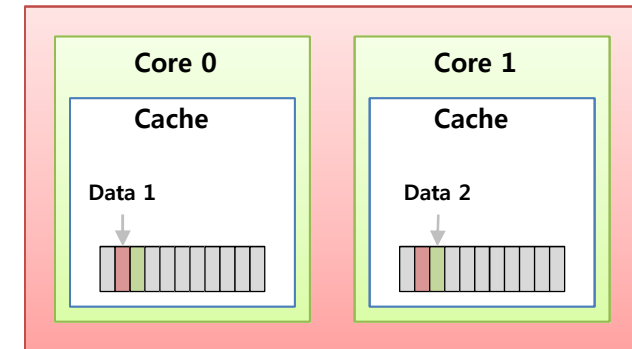
- Cache Line이 어떤 한 Processor에 의해 고쳐졌음 (Dirty)

Cache

False Sharing

■ False Sharing Example

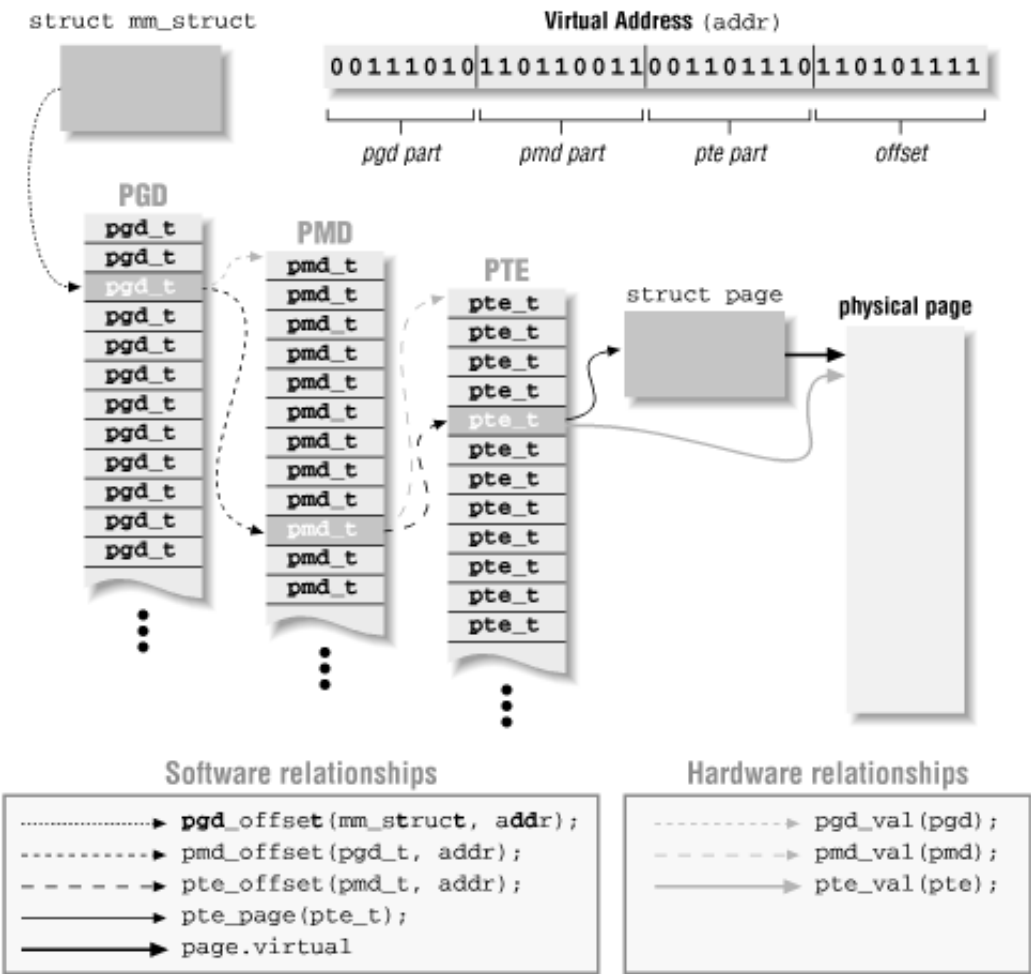
- 0번 Processor가 Data 1을, 1번 Processor가 Data 2를 갱신한다.
 - 각 Core는 전용 L1 Cache (Private Cache)를 가지고 있다.
 - Cache를 읽고 쓰는 단위는 64Byte와 같이 Cache Line 만큼 처리된다.
 - Core 0의 전용 Cache에는 Core 0가 갱신하는 Data 1이 있음.
 - Data 2는 Core 1의 전용 Cache에 존재한다.
 - Locality에 의해 Data 1과 Data 2가 동일 Cache Line에 위치할 확률이 높다.
 - Data 1과 Data 2가 연이어 선언되어 있다 (그림참조).
 - Core 0가 Data 1을 갱신할 때 다른 Core가 가지고 있을지도 모르는 이 Cache Line을 무효화 시켜야 한다.
 - Core 1도 Data 2를 갱신할때 이 Cache Line을 무효화 시켜야 한다.
-
- Core 0이 Data 1이 있는 Cache Line을 무효화시키면 Core 1은 Data 2가 있는 Cache Line을 잃어버리게 된다.
 - Core1이 Data 2를 읽을 때 Cache Miss가 발생하게 된다. (Coherence Miss)
 - 실제로 공유되지 않은 Data가 없음에도 불구하고 끊임없이 서로의 Cache Line을 무효화하여 Cache Miss를 유발시킨다.



___cacheline_aligned을 통해 cache line 정렬 (빈공간 Padding)을 통해 해결

```
spinlock_t async_lock ___cacheline_aligned;  
int req ___cacheline_aligned;      /* transmit slots submitted */  
int done ___cacheline_aligned;     /* transmit slots completed */
```

PGD, PMD, PTE 관계 (커널에서)



Preemption

■ Background

- Kernel preemption을 통해 urgent process를 normal process 보다 빠르게 수행할 수 있음.
- Kernel preemption은 2.5에서 추가됨
- Kernel preemption은 User land의 preemption과 다른 Concept.

■ SMP Code 차용

- Kernel preemption을 지원하기위해서 Kernel이 약간 수정되었지만, user land preemption 처럼 쉬운것은 아니었다. 커널이 single operation에서 certain action을 complete하지 못한다면 (data manipulation) race condition이 발생함. Multiprocessor system에서 발생하는 동일한 문제임. 이러한 문제점은 모두 SMP에서 모두 identify 되었음. SMP implementation에서 적용되어 재사용 가능.
- Kernel이 선점될 수 있다면, uniprocessor system도 SMP와 같이 동작함

■ Preemption 관련 보조함수

- preempt_disable --> inc_preempt_count, memory optimization 회피 (kernel preemption mechanism에 문제 야기할 수 있기 때문)
- preempt_check_resched --> scheduling이 필요한지 검사하고 필요하며 scheduling
- preempt_enable --> enable kernel preemption --> preempt_check_reched
- preempt_disable_no_resched --> disabling preemption (no rescheduling)
- SMP system을 위해서 dec_preempt_count/ inc_preempt_count는 synchronization operation에 통합됨
- get_cpu, put_cpu는 kernel preemption을 disable 시킨다.

■ Preemption Checking

- struct thread_info의 preempt_count를 이용해서 preempt될 수 있는지 tracking
- preempt_count = 0 --> preemptable, >0 --> unpreemptable
- preempt_count는 dec_preempt_count, inc_preempt_count를 통해서 값 변경
- preempt_count를 Boolean value로 하지 않는 것은 critical section에 다양한 route로 접근가능하기 때문

Preemption

■ SMP vs Kernel preemption의 차이

- SMP에서는 per-CPU 관련 변수에 대해 protection을 할 필요가 없었지만, kernel preemption에서는 고려해야함. 왜냐하면 같은 CPU에서 다른 route를 통해서 그 값을 참조할 수 있기 때문임

■ preempt count vs. PREEMPT_ACTIVE

- preempt_count는 다른 task가 kernel을 선점할 수 있는지 검사하기 위한 변수
- 반면 PREEMPT_ACTIVE는 scheduler가 scheduling 호출이 선점에 의해 호출되었는지를 검사하기 위한 flag

■ How does the kernel know if preemption is required?

- preempt_enable -->
- preempt_check_reched --> test_thread_flag (TIF_NEED_RESCHED) -->
- TIF_NEED_RESCHED flag --> CPU time을 요청한 process가 있음을 의미 -->
- preempt_schedule : preempt_count > 0, irqs_disabled되었으면 just return
- add_preempt_count(PREEMPT_ACTIVE) : scheduling요청이 preemption으로 부터왔음을 마킹
- schedule : 선점에 의한 스케줄링 요청일경우 deactivate_task를 건너뛰고 high-priority task가 스케줄됨

■ kernel preemption이 trigerring되는 2가지 방법

- preempt_schedule
- preempt_schedule_irq

* preemption request가 IRQ context로 부터 왔음을 의미

* IRQ가 disable된 상태에서 call됨 (Simultaneous IRQ에 의한 recursive call을 방지하기 위함)

■ Low Latency

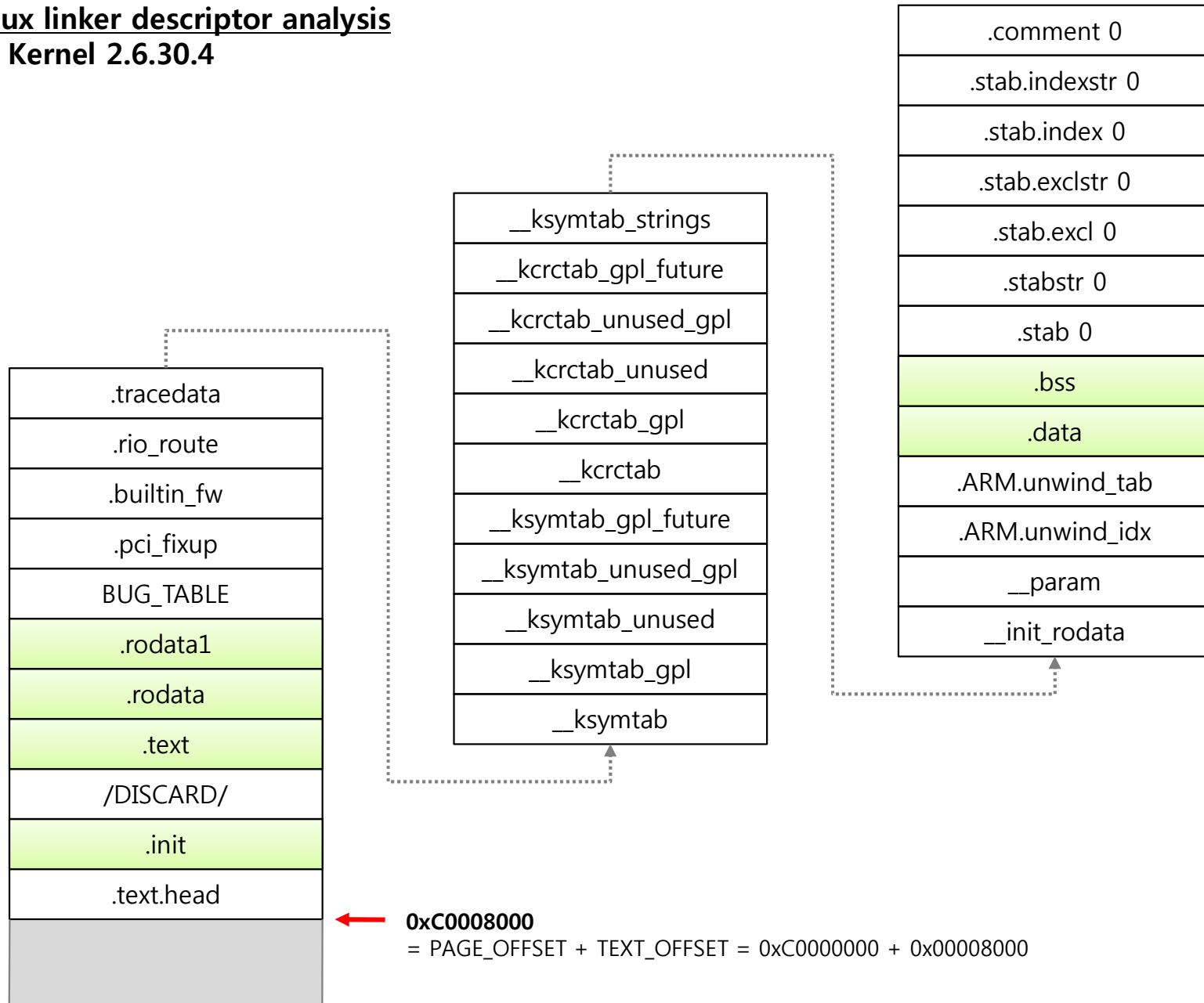
- cond_resched --> long operation 중간중간에 call되어 다른 process가 동작할 수 있도록 해줌. preemption과 관계 없음

Kernel Locks

Lock Types	Locks	Description
Sleep 할 수 있는 Lock	Semaphore, Mutex	
	Reader Writer Semaphore (rwsem)	Read를 위해서 Multiple Access가 가능하나 Write를 위해서는 한 개의 Process만 진입가능
	Completion	Semaphore의 경우 up/down이 빈번히 발생하면 up 연산이 일어나기 전에 Semaphore 자료구조가 기다리지 않고 제거될 수 있음. Semaphore의 up()은 complete()로, down()은 wait_for_completion()으로 대체
Sleep 할 수 없는 Lock	Spinlocks	Lock이 다른 process에 의해 획득이되었다면, Lock이 풀릴때까지 Spin. SMP상에서 Locking을 구현하기위한 방법임
	Seqlocks	Lock없이 공유자원에 접근함. 읽기 작업의 경우 바로 접근하지만, 쓰기 작업과 충돌이 날 경우 읽기작업에서 재시도를 통해 값을 읽어옴
기타	RCU (Read-Copy-Update)	Spinlock의 비효율성을 개선하기위해 나온것이며, 수정이 필요한 Writer는 해당 데이터를 복사하고 변경한뒤 Reader가 읽고 있는 데이터가 정리되면 포인터만 변경함
	Atomic Variables	공유자원이 단순한 정수값일 경우 Lock을 이용하는 대신, atomic_t 구조체 타입을 이용하고 단일 기계어로 컴파일 되게 하여 실행속도를 빠르게 함.
	BIT Operation	원자적으로 개별 bit를 설정할 경우. 예로, set_bit()

Linker Descriptor

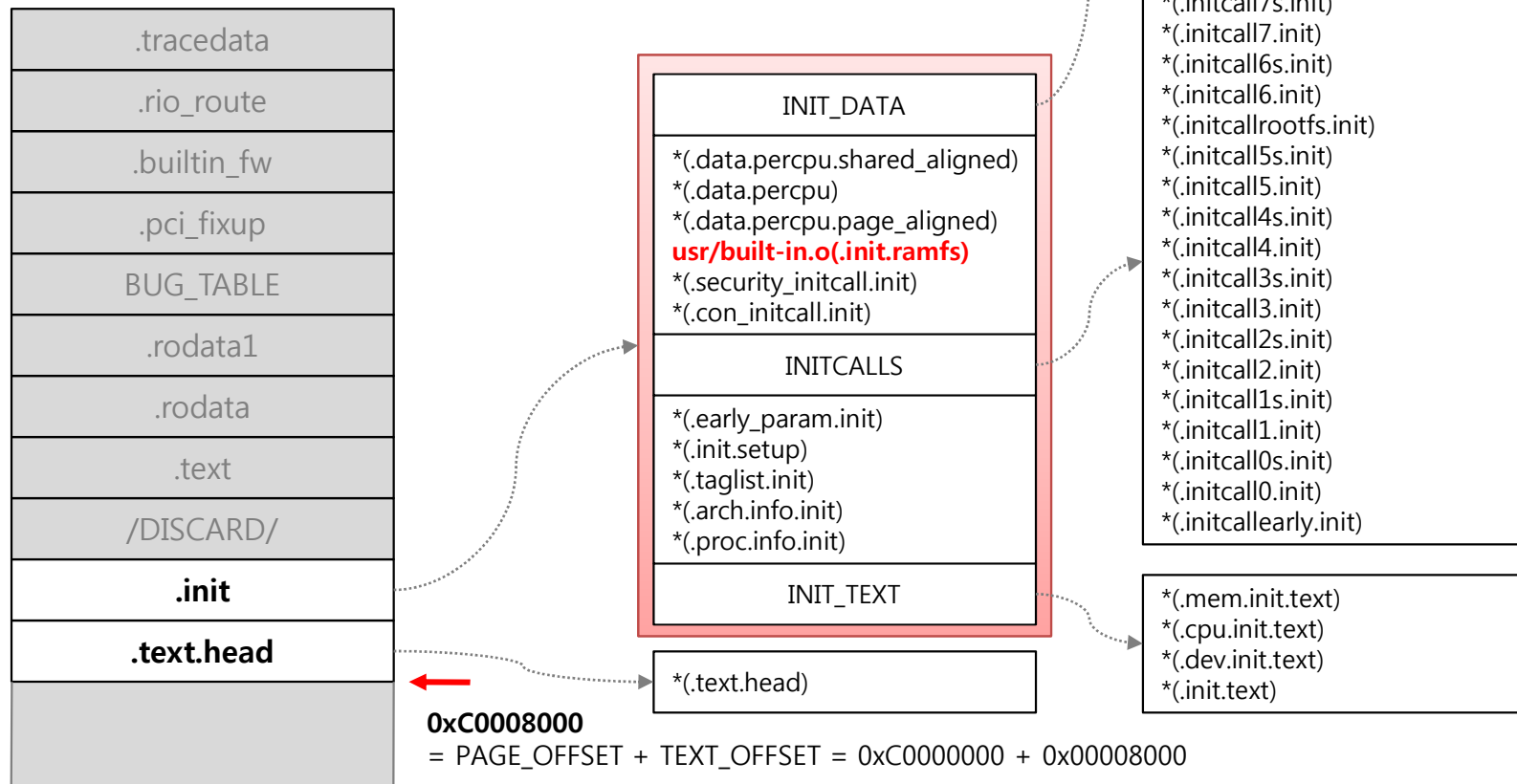
vmlinux linker descriptor analysis Linux Kernel 2.6.30.4



vmlinux linker descriptor analysis Linux Kernel 2.6.30.4

```
*(.dev.*) = if !CONFIG_HOTPLUG
*(.cpu.*) = if !CONFIG_HOTPLUG_CPU
*(.mem.*) = if !CONFIG_MEMORY_HOTPLUG
```

```
usr/built-in.o(.init.ramfs) = if CONFIG_BLK_DEV_INITRD
```



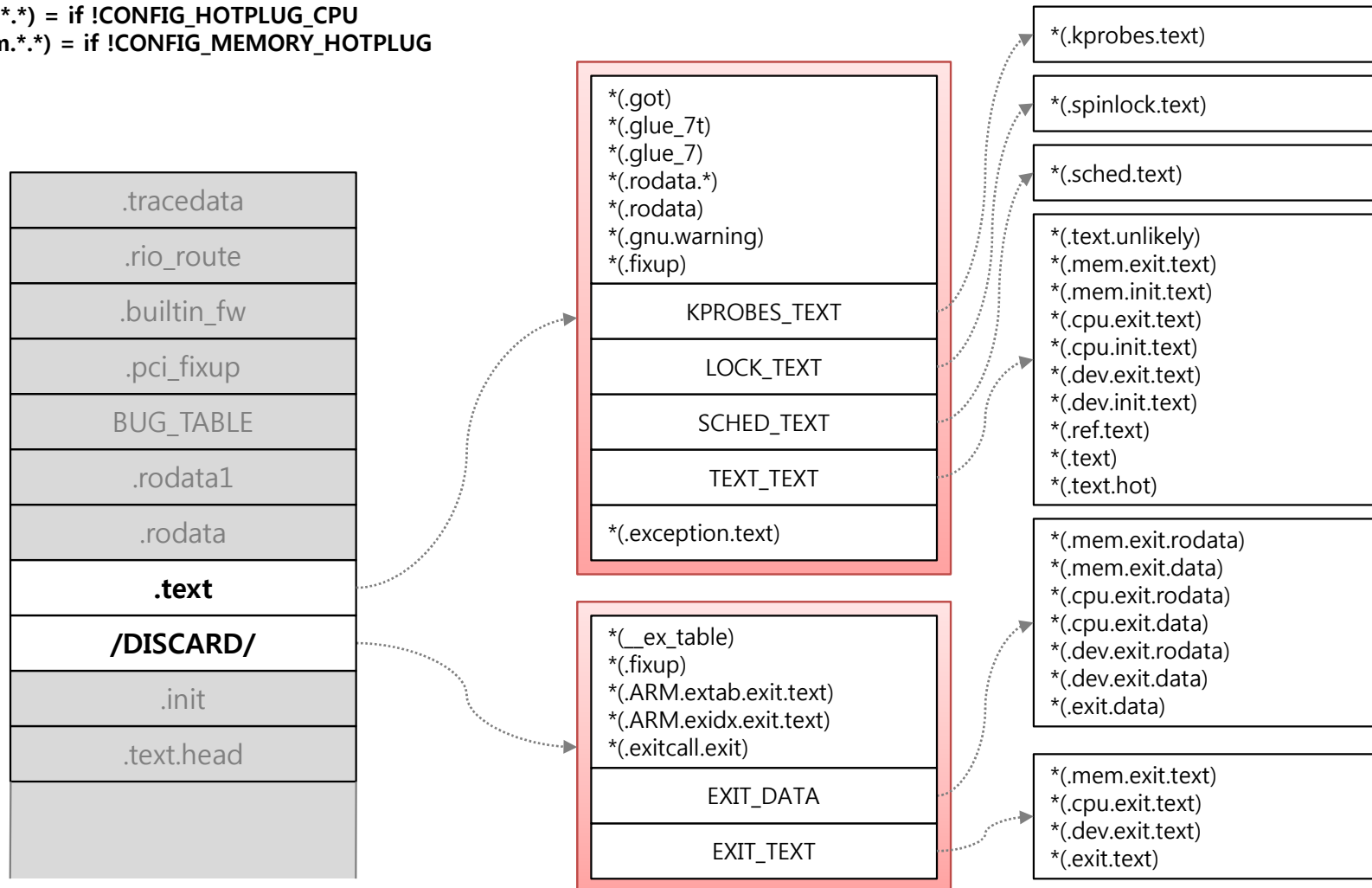
vmlinux linker descriptor analysis

Linux Kernel 2.6.30.4

```

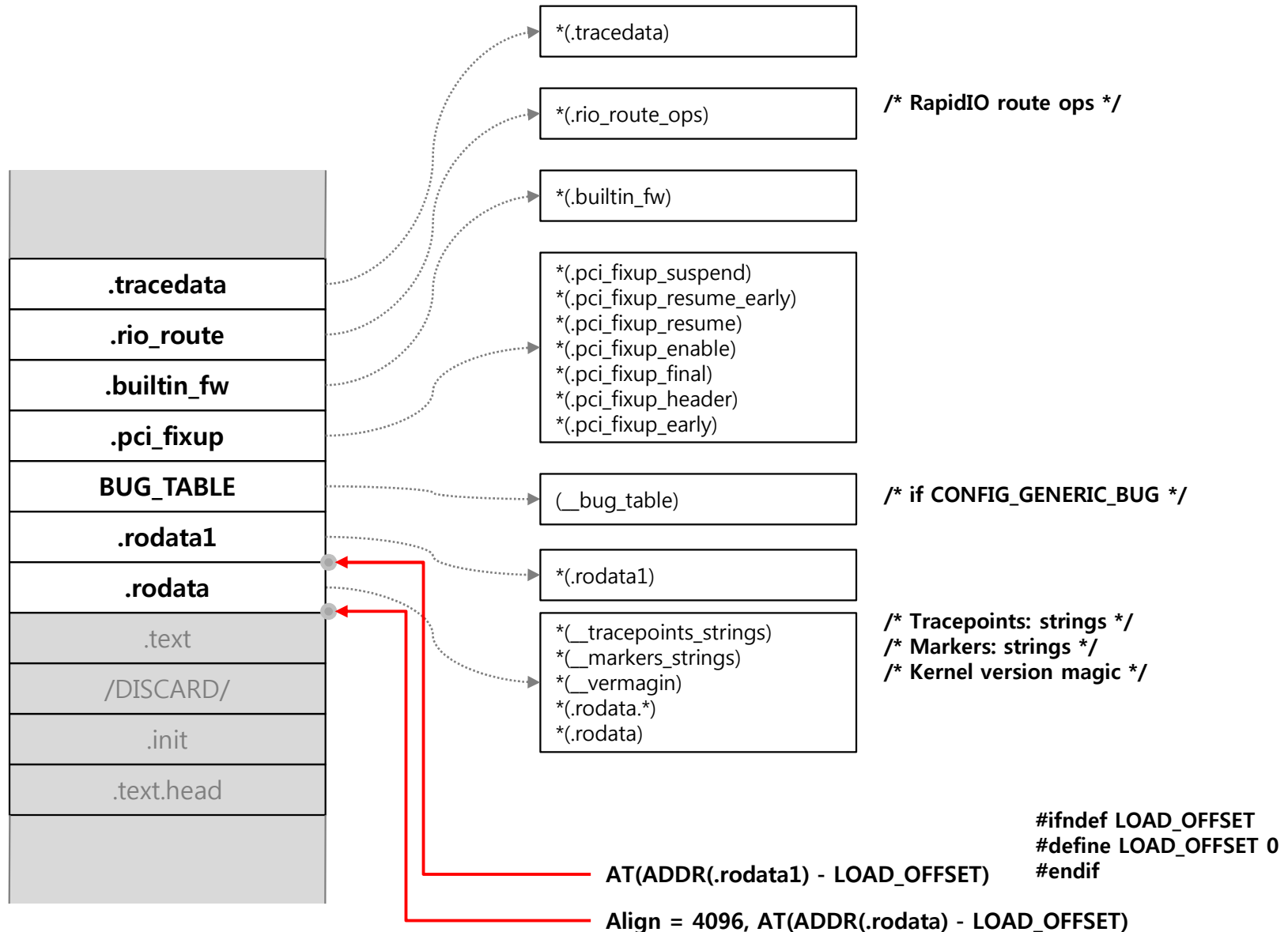
*(__ex_table) = if CONFIG_MMU
*(.fixup) = if CONFIG_MMU
*(.dev.*) = if !CONFIG_HOTPLUG
*(.cpu.*) = if !CONFIG_HOTPLUG_CPU
*(.mem.*) = if !CONFIG_MEMORY_HOTPLUG

```



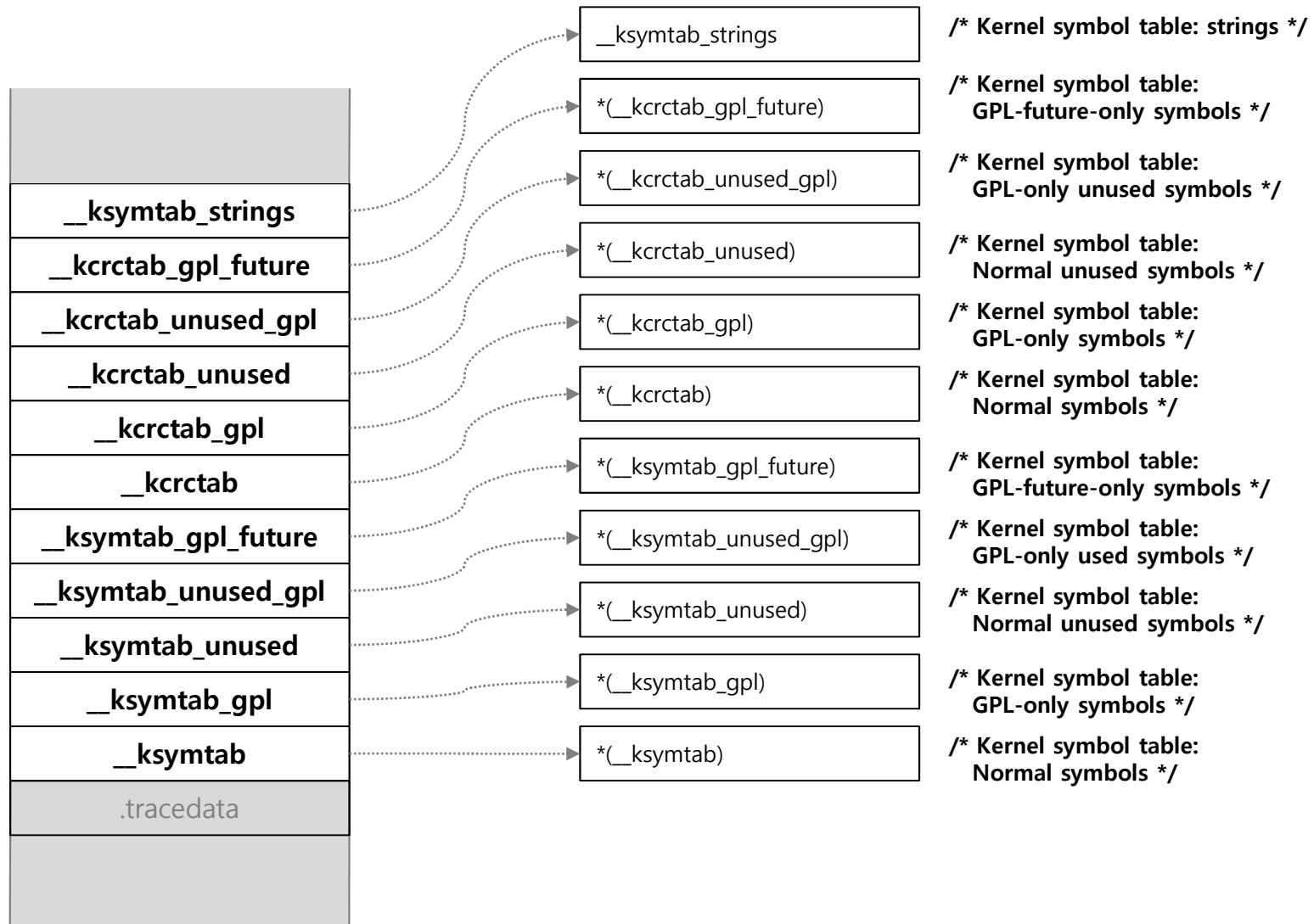
vmlinux linker descriptor analysis

Linux Kernel 2.6.30.4



vmlinux linker descriptor analysis

Linux Kernel 2.6.30.4



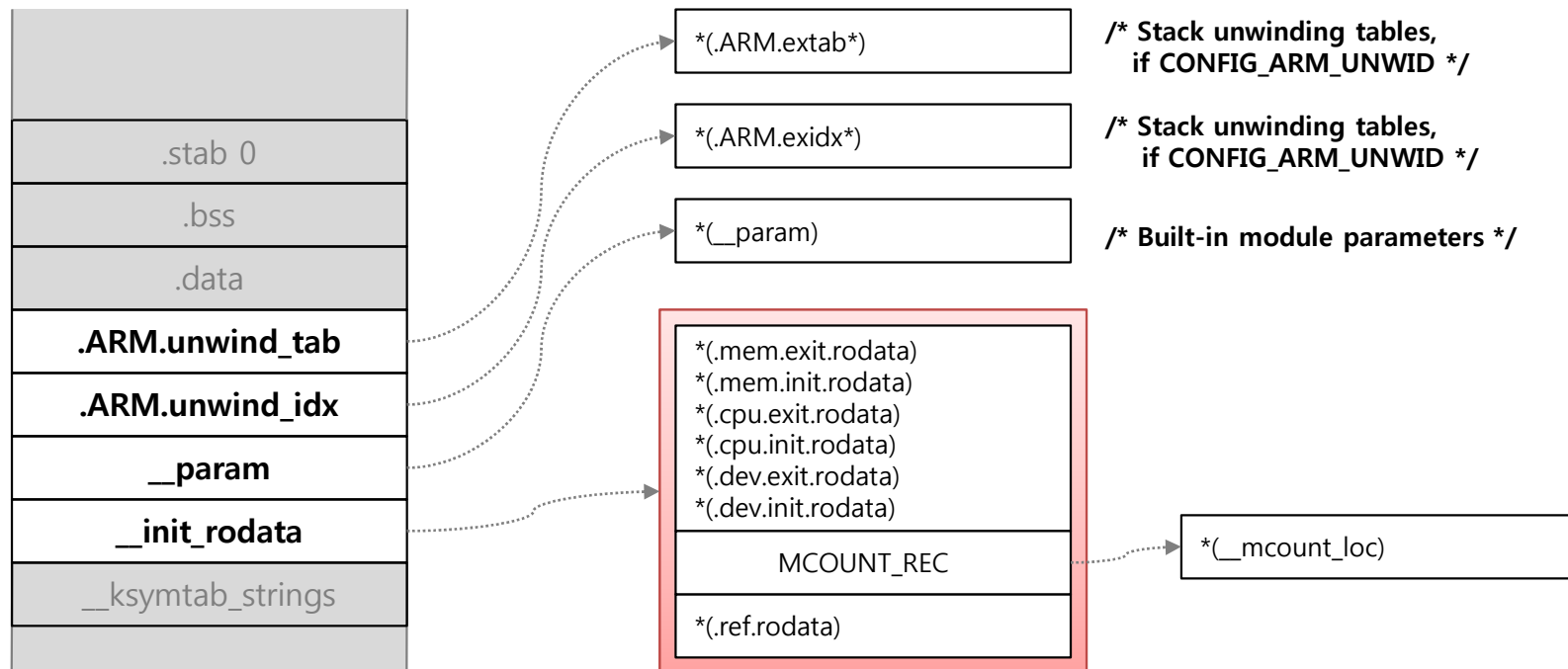
vmlinux linker descriptor analysis

Linux Kernel 2.6.30.4

```

*(__mcount_loc) = if CONFIG_FTRACE_MCOUNT_RECORD
*(.dev.*.*) = if !CONFIG_HOTPLUG
*(.cpu.*.*) = if !CONFIG_HOTPLUG_CPU
*(.mem.*.*) = if !CONFIG_MEMORY_HOTPLUG

```

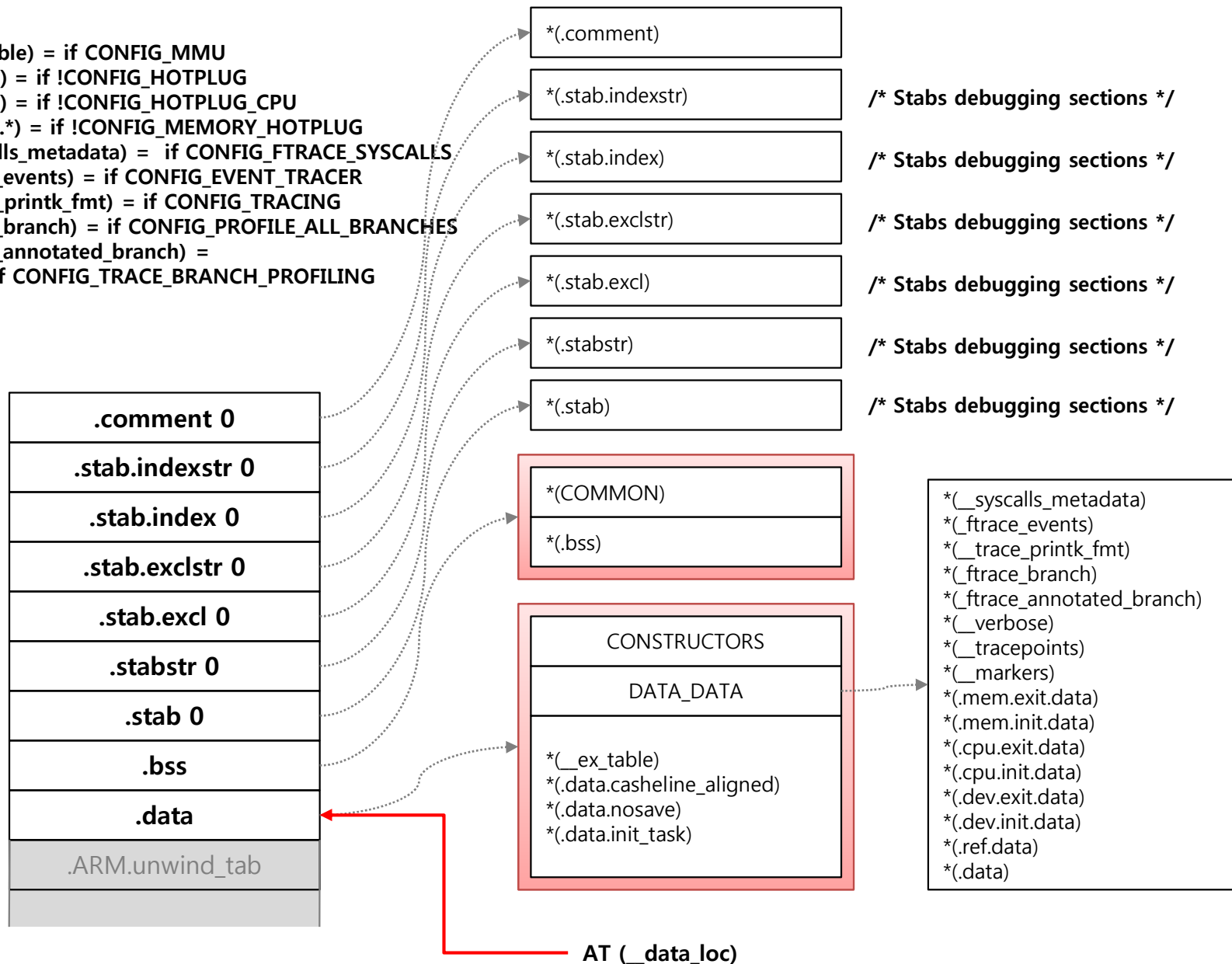


vmlinux linker descriptor analysis Linux Kernel 2.6.30.4

```

*(__ex_table) = if CONFIG_MMU
*(.dev.*.*) = if !CONFIG_HOTPLUG
*(.cpu.*.*) = if !CONFIG_HOTPLUG_CPU
*(.mem.*.*) = if !CONFIG_MEMORY_HOTPLUG
*(__syscalls_metadata) = if CONFIG_FTRACE_SYSCALLS
*(ftrace_events) = if CONFIG_EVENT_TRACER
*(__trace_printk_fmt) = if CONFIG_TRACING
*(ftrace_branch) = if CONFIG_PROFILE_ALL_BRANCHES
*(ftrace_annotated_branch) =
    if CONFIG_TRACE_BRANCH_PROFILING

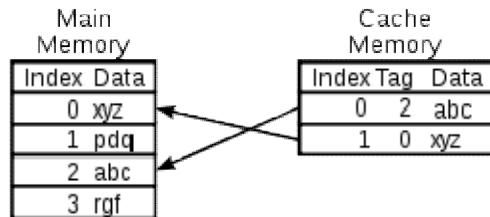
```



Cache Type

Cache Type: PIPT, VIVT, VIPT, PIVT

Cache Type은 index와 tag가 physical address로 표현되는지 virtual address로 표현되는지에 따라 4가지로 구분된다. index & tag에 대한 개념은 ARM System Developer's Guide의 460 page 참조.

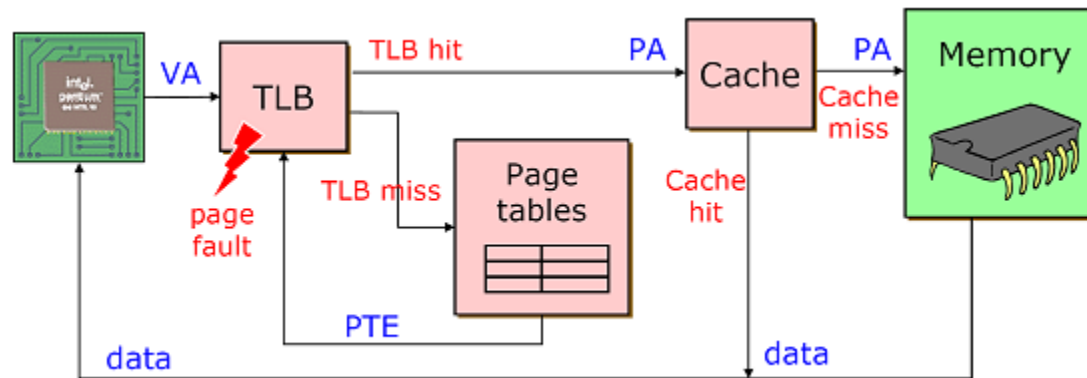


Cache 에 Tag가 있는데, 이게 Main Memory의 위치를 가르키는 값을 포함합니다. 이 값이 virtual이냐 physical이냐에 따라 virtually tagged 나 physically tagged냐로 구분할 수 있습니다. Index라 함은 processor가 참조하는 주소가 되는데, virtual이냐 physical이냐로 나뉩니다. physical이 될려면 일단 MMU가 개입이 되어야 합니다 (MMU를 이용하는 Linux라 가정함).

source: <http://upload.wikimedia.org/wikipedia/commons/3/3d/Cache%2Cbasic.svg>

PIPT (Physically indexed, physically tagged)

index와 tag에 모두 physical address가 사용됨. 가장 간단한 방법이고 physical address는 유일하기 때문에 address aliasing 문제가 없지만, physical address로 변경이 필요하게 됨. 그 의미는 TLB의 개입이 필요하고, TLB miss가 발생할 시에 속도가 떨어짐. TLB miss가 발생하지 않는다 하더라도, TLB lookup 후에 Cache lookup을 sequential하게 이루어져야 함. 따라서 전체적으로 속도가 slow.

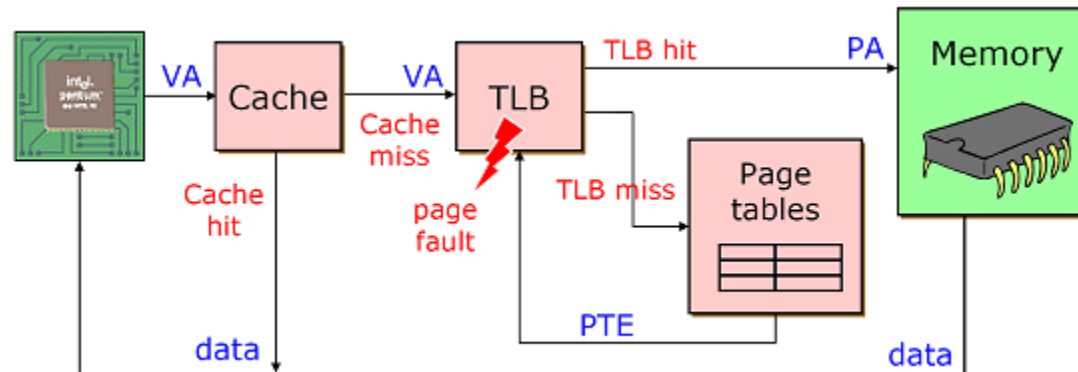


source: <http://arina319.tistory.com/archive/20090107>

Cache Type: PIPT, VIVT, VIPT, PIVT

VIVT (Virtually indexed, virtually tagged)

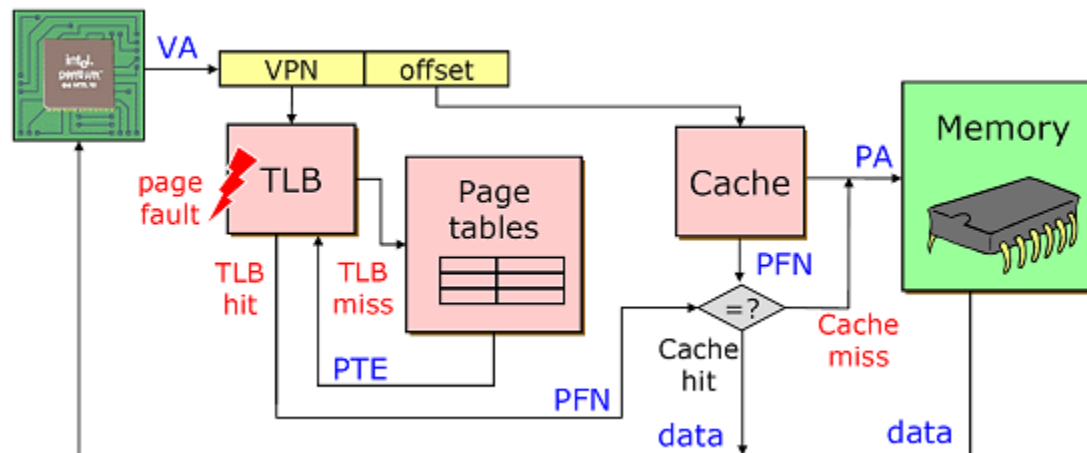
index와 tag에 모두 virtual address가 사용됨. virtual address가 사용되기 때문에 physical address를 찾기 위해 MMU가 개입될 필요가 없음. 하지만, 서로다른 virtual addresses가 같은 physical address를 가리키는 aliasing 문제가 발생함. 이는 physical address에 서로다른 virtual address가 cache될 때 발생하는데, coherency problem을 발생시킴. 예를 들면, 서로다른 사용자 application이 동일한 file을 mmap할 때 사용자 application의 virtual address는 다르나 실제 physical address는 같게됨. 다른 문제점으로는 V->P mapping (virtual to physical)이 바뀔 수 있다는 점이며, 이경우 TLB entry의 변경에 주의해야하고 entry가 변경되기 전에 해당 cache line들을 flushing해야 함. 왜냐하면, virtual address가 더이상 유효하지 않기 때문에.



Cache Type: PIPT, VIVT, VIPT, PIVT

VIPT (Virtually indexed, physically tagged)

index에는 virtual address를 사용하고 tag에는 physical address를 사용함. PIPT 대비 low latency (physical address를 찾는데 걸리는 시간)의 장점이 있음. TLB를 통해 cache line을 parallel 하게 찾을 수 있음. 하지만, physical address를 찾을 때까지 tag를 비교할 수는 없음. 왜냐하면 tag가 physical address를 사용하기 때문에, processor가 참조하는 Index는 virtual address이기 때문에 이를 physical address로 변경이 필요하다는 의미임. VIVT 대비 장점은 tag가 physical address이기 때문에 cache가 aliasing을 검출할 수 있음. VIPT는 약간의 tag bit가 좀 더 필요한데, 왜냐하면 index bit가 더이상 동일한 address를 표현하지 않기 때문임.



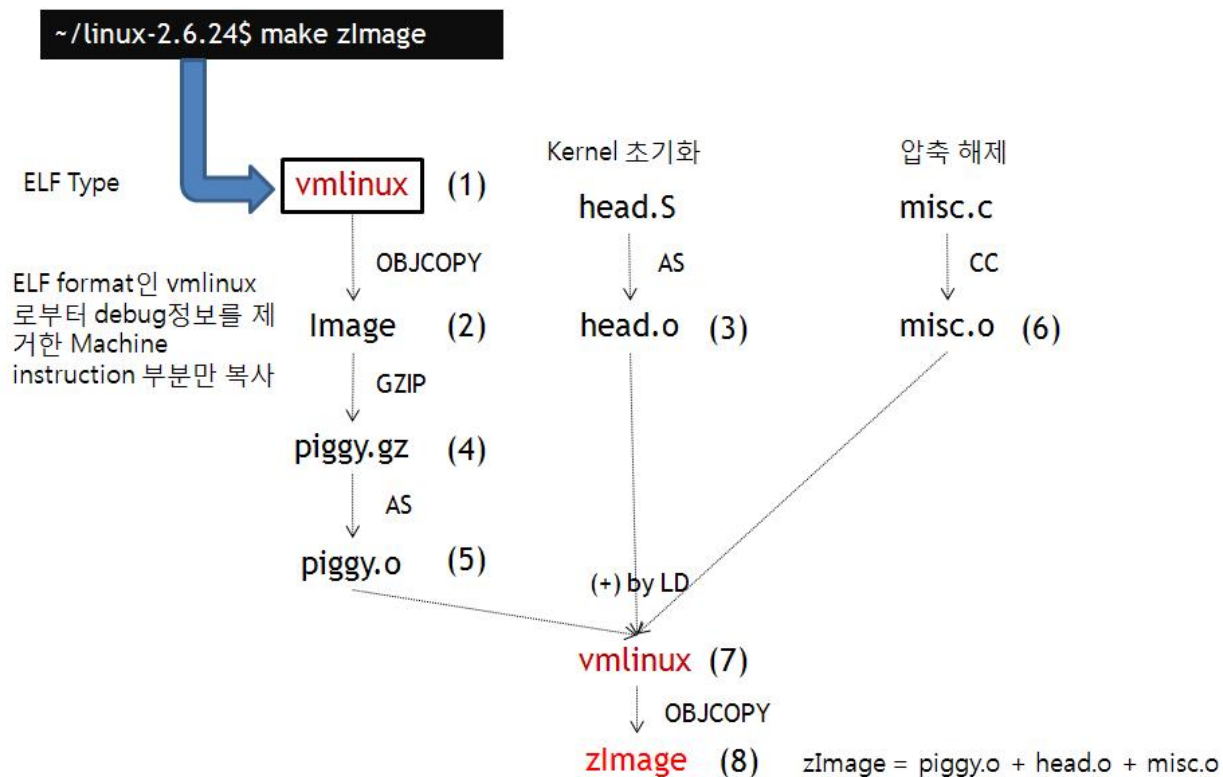
PIVT (Physically indexed, virtually tagged)

이론적으로만 존재할 뿐 useless한 type

Tips

Kernel을 컴파일 하면 나오는 파일은 vmlinux입니다. file vmlinux 라고 리눅스 명령을 내리면 vmlinux file type이 ELF라는 것을 알 수 있습니다. vmlinux에는 디버깅용부터 추가적인 정보를 포함하고 있습니다. objcopy를 통해 vmlinux에서 순수하게 Instruction set만 뽑아낸 것이 Image file입니다. 이 Image file은 메모리에 로드한 후 PC (program counter)를 Image가 로드된곳으로 이동시키면 linux가 부팅이 되는 이미지 입니다. 그런데 이 Image의 크기가 좀 큼니다. 따라서, 그 다음 스텝이 이 Image를 gzip을 통해서 압축을 합니다. 그러면 piggy.gz이 나오게 되고 Head.S에서 piggy.gz를 데이터로 로드해서 piggy.o를 만듭니다. piggy.o가 압축된 파일이기 때문에 이를 풀어줄 뭔가가 필요하게 되죠. 그게 바로 misc.o입니다. 그러면 여기서 왜 Image 파일을 압축을 하는가? 그 이유는 Image file을 Embedded 장비의 플래시 메모리에 적재시킨 후 메모리로 복사하는 시간이 압축된 Image 파일을 메모리에 복사하고 메모리에서 압축을 해제한 후 수행 시키는 시간이 더 적게걸리기 때문입니다.

아래 이미지는 make zImage 했을때 수행되는 순서입니다.



Debugging용 Profiling Code 추가

```
#define __USE_GNU
#include <dlfcn.h>
```

__no_instrument_function__은 이 함수가 profiling되지 않도록 함. Gprof을 수행시킬때 이 옵션이 없으면 stack overflow 발생

```
void __attribute__((__no_instrument_function__)) __cyg_profile_func_enter(void *this_fn, void *call_site)
{
    Dl_info info_this;
    Dl_info info_callor;

    dladdr(this_fn, &info_this);
    dladdr(call_site, &info_callor);
}
```

```
void __attribute__((__no_instrument_function__)) __cyg_profile_func_exit(void *this_fn, void *call_site)
{
    Dl_info info_this;
    Dl_info info_callor;

    dladdr(this_fn, &info_this);
    dladdr(call_site, &info_callor);
}
```

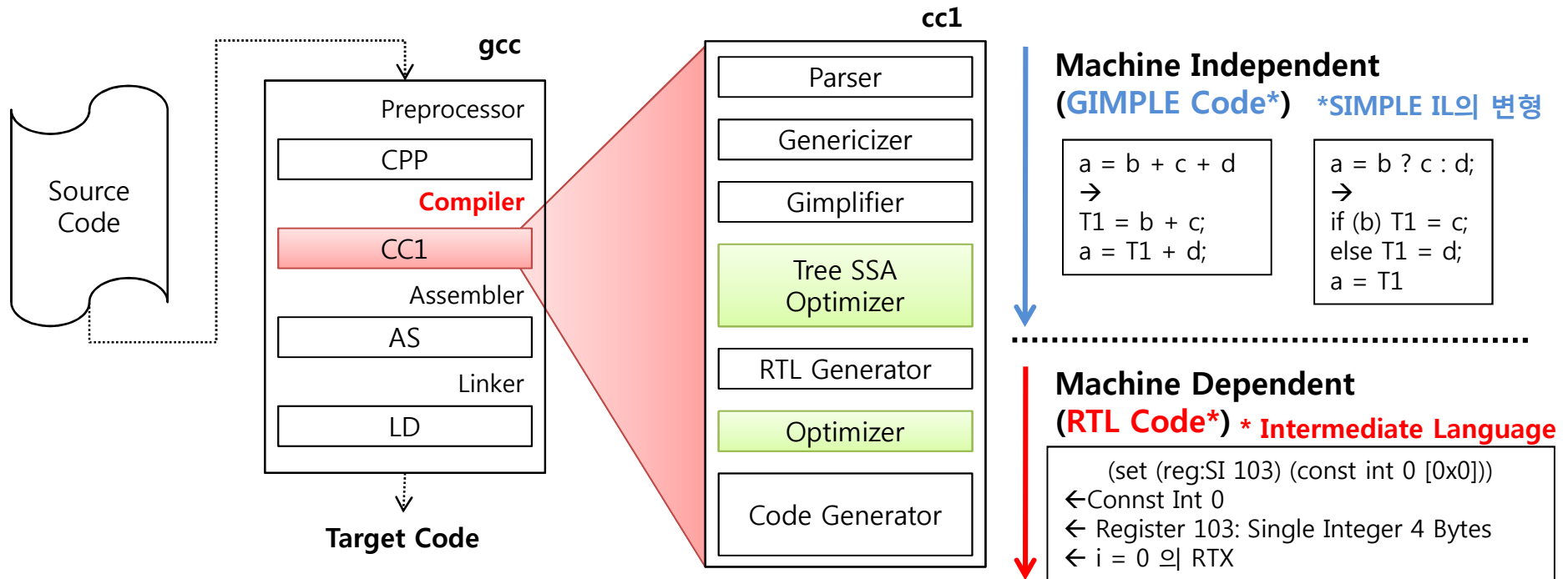
To Do
Function Pointer로 부터 Function Name을 알아내는 방법은? Address를 보여주는 것은 Debugging에 별 도움이 안됨.

```
$ gcc -finstrument-functions -ldl -g xxxx.c -o xxxx
```

-finstrument-functions 옵션을 사용하면, 함수의 시작과 끝에 profile 함수를 추가함.
-ldl 옵션으로 dladdr 함수를 사용할 수 있음

GCC Concept

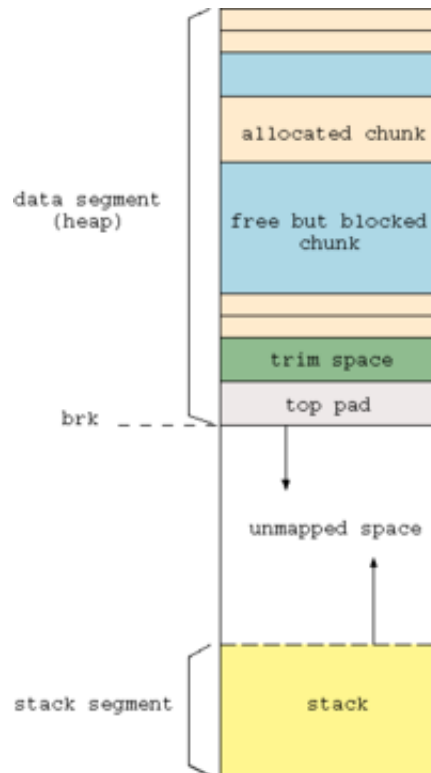
GCC는 GNU Compiler Collection의 약자이며, 이 Collection에는 g++ (GNU C++), gcc (GNU C), Fortran, Java등의 Compiler를 포함하고 있다. (over 280만 LOC)



- **Portable Optimizer (PO)**에 사용된 RTL이 gcc에서 PO를 적용하면서 사용됨
- GCC 4.x에서 RTL보다는 GIMPLE에 비중을 높임
- RTL을 점점 제거하는 추세

- **Parser**: Parse Tree를 만들어 Grammatical Check
- **Genericizer**: 언어독립 Tree 구성 (GENERIC)
- **Gimplifier**: GENERIC → GIMPLE로 변경
- **SSA**: Single Static Assignment로 모든 변수가 한번 assign됨
- **RTL Generator**: Register Transfer Language로 GIMPLE → RTL 변환
- **Optimizer**: RTL Code를 Optimization
- **Code Generator**: Assembly Language로 생성

Question: 사용자가 malloc call시 Kernel의 메모리 할당자가 메모리를 할당하는가?



*malloc은 dmalloc (Doug Lea Malloc)을 통해 구현되어 있음. Doug Lea가 구현한 것으로 많은 malloc() 함수가 Doug Lea의 알고리즘을 따른다고 함. malloc은 유저공간에서 메모리 할당을 해주는 메모리 할당자임.

■ Kernel의 메모리 할당:

- ➔ Buddy Allocator 및 Slab을 통해 처리됨. **커널 내부적으로 물리적으로 연속된 메모리가 필요할 때 사용함**

■ malloc시 메모리 할당:

- ➔ heap을 이용하는 방법: heap은 brk를 통해서 미리 할당 받아 놓고 이를 조각내어 유저공간에 제공함. 옆 그림에서 brk가 가르키고 있는 부분 (약 128KB)
- ➔ mmap을 이용하는 방법: 1MB이상과 같이 큰 메모리를 할당할 때는 mmap을 통해서 메모리를 할당받음. 이때는 4KB단위로 할당됨.

■ mmap vs. brk의 차이점?

- ➔ brk는 Process의 가상공간에서 bss가 위치한 바로 위로 부터 특정 크기를 할당함. 이렇게 되면 항상 Base주소는 같게 되고 **brk를 통해 그 위치만 Update**. allocated chunk하고 free but blocked가 된 부분을 볼 수 있음 (옆 그림). 이는 메모리 영역을 잘게 나뉘서 필요한 만큼을 애플리케이션으로 전달함
- ➔ mmap은 말 그대로 unmapped space에서 필요한 만큼의 물리메모리를 가상공간으로 매핑해서 전달해 주는 것임. 이렇게 되면 mmap, munmap시 Fragmentation이 발생하게되고 주소가 매번 Update되어야 하는 문제가 있음.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int *p;
    p = (int*)malloc(1024*atoi(argv[1]));
    p[1] = 1024;
    printf("the value = %d, %p\n", p[1], p);
    printf("[0]=%p, [1]=%p, ... i[9]= %p\n", &p[0], &p[1], &p[9]);
    free(p);
}
```

<malloc_test_with_num.c>

```
$strace ./malloc_test_with_num 127
$strace ./malloc_test_with_num 400
```

그림소스: <http://www.linuxjournal.com/article/6390>

Program Execution Steps

[source: m.c]

```
extern out_func(char* str);
int main(int argc, char* argv[])
{
    out_func("hello linker & loader\n");
    return 0;
}
```

[source: func.c]

```
#include <stdio.h>

void out_func(char* str)
{
    printf("%s", str);
}
```

[Compile & Linking]

```
$ gcc -c m.c func.c
$ gcc -o out m.o func.o
```

```
rsyoung@diogenes:~/prgtest/linker_loader/ex2$ ls
func.c func.o m.c m.o out
rsyoung@diogenes:~/prgtest/linker_loader/ex2$ objdump -f m.o

m.o:          file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

rsyoung@diogenes:~/prgtest/linker_loader/ex2$ objdump -f func.o

func.o:       file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

rsyoung@diogenes:~/prgtest/linker_loader/ex2$ objdump -f out

out:         file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482f0
```

Linker에 의한
재배치

[Deassembling]

```
$ objdump -d out
```

```
Disassembly of section .text:

080482f0 <start>:
80482f0: 31 ed                xor     %ebp,%ebp
80482f2: 5e                   pop     %esi
80482f3: 89 e1                mov     %esp,%ecx
80482f5: 83 e4 f0             and     $0xffffffff0,%esp
80482f8: 50                   push    %eax
80482f9: 54                   push    %esp
80482fa: 52                   push    %edx
80482fb: 68 c0 83 04 08       push    $0x80483c0
8048300: 68 d0 83 04 08       push    $0x80483d0
8048305: 51                   push    %ecx
8048306: 56                   push    %esi
8048307: 68 74 83 04 08       push    $0x8048374
804830c: e8 b7 ff ff ff       call    80482c8 <libc_start_main@plt>
8048311: f4                   hlt
8048312: 90                   nop
```

```
080482c8 <libc_start_main@plt>:
80482c8: ff 25 98 95 04 08    jmp     *0x8049598
80482ce: 68 08 00 00 00       push    $0x8
80482d3: e9 d0 ff ff ff       jmp     80482a8 <_init+0x30>
```

```
$ objdump -R out
```

Relocation Table을 보여줌

```
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049584 R_386_GLOB_DAT __gmon_start__
08049594 R_386_JUMP_SLOT __gmon_start__
08049598 R_386_JUMP_SLOT libc_start_main
0804959c R_386_JUMP_SLOT printf
```

<main function>

```
08048374 <main>:
8048374: 8d 4c 24 04          lea     0x4(%esp),%ecx
8048378: 83 e4 f0             and     $0xffffffff0,%esp
804837b: ff 71 fc             pushl   -0x4(%ecx)
804837e: 55                   push    %ebp
804837f: 89 e5                mov     %esp,%ebp
```

<out_func relocation>function>

```
080483a0 <out_func>:
80483a0: 55                   push    %ebp
80483a1: 89 e5                mov     %esp,%ebp
80483a3: 83 ec 08             sub     $0x8,%esp
```

Reference

To be updated

1. Wolfgang Mauerer, Professional Linux Kernel Analysis – Wrox
2. 백승재외, 리눅스 커널 내부구조 – 교학사
3. arina02.tistory.com
4. 김민장, 프로그래머가 몰랐던 멀티코어 CPU 이야기
5. 정준석, IT EXPERT 임베디드 개발자를 위한 파일시스템의 원리와 실습
6. Kurt Wall, GCC 완전정복
7. 리눅스 커널 2.6 구조와 원리, 한빛미디어
8. Embedded Linux Device Driver – MSD 아카데미
9. Cache Type : <http://arina319.tistory.com/archive/20090107>



Epilogue

“ 분석했던 코드는 잊을 수도 있겠지만
함께했던 여러분들은 잊지 못할 겁니다 ”

iamroot 어느 멤버가...