Android Kernel Hacks

안드로이드를 위한 리눅스 커널 핵스

2013.7.16 ~ 2013.9.27

이 충 한(chunghan.yi@gmail.com, slowboot)

목차

CHAPTER 3

주요 Kernel 프로그래밍 기법 2

본 장에서는 안드로이드의 핵심인 리눅스 커널을 소개하고, 커널 프로그래머가 놓치기 쉬운 커널 프로그래밍 기법에 대하여 소개하고자 한다.

- Kernel 초기화 과정 분석

- Task, Wait Queue, Scheduling, 그리고 Preemption
- Top Half, Bottom Halves와 Deferring Work
  - Hard Interrupt Handler, Softirq, Tasklet, Work Queue, Interrupt Thread
- Timer

- 동기화 방식(Synchronization Method)
  - Spinlock, Mutex, Semaphore, Completion, RCU
- Notifier Chains
- 메모리 관련 프로그래밍 기법
- Device Model & Sysfs

1. 동기화(Synchronization)

동기화 상황과 주의 사항

Linux는 아래와 같은 concurrency 상황이 발생할 수 있으며, 동시에 실행 가능한 상황에 처해 있는 코드는 적절히 보호되어야 한다.

Interrupts

- *interrupt는 아무 때나 발생(asynchronously)하여, 현재 실행중인 코드를 중단시킬 수 있다.*

Softirqs & tasklet

- *얘들은 kernel이 발생시키고, schedule하게 되는데, 얘들도 거의 아무때나 발생하여 현재 실행중인 코드를 중단시킬 수 있다.*

Kernel preemption

- *글자 그대로 한 개의 task가 사용하던 CPU를 다른 task가 선점(CPU를 차지)할 수 있다 (linux 2.6 부터는 fully preemptive).*
- *<Kernel preemption이 발생하는 경우>*
  - *Interrupt handler가 끝나고, kernel space로 돌아갈 때*
  - *Kernel code가 다시 preemptible해 질 때(코드 상에서)*
  - *Kernel task가 schedule( ) 함수를 호출할 때*
  - *Kernel task가 block될 때(결국은 schedule( ) 함수를 호출하는 결과 초래)*

Sleeping 및 user space와의 동기화

- *kernel task는 sleep할 수 있으며, 그 사이 user-process(system call)가 CPU를 차지할 수 있다.*

Symmetrical multiprocessing

- *두 개 이상의 processor(CPU)가 동시에 같은 kernel code를 실행할 수 있다.*

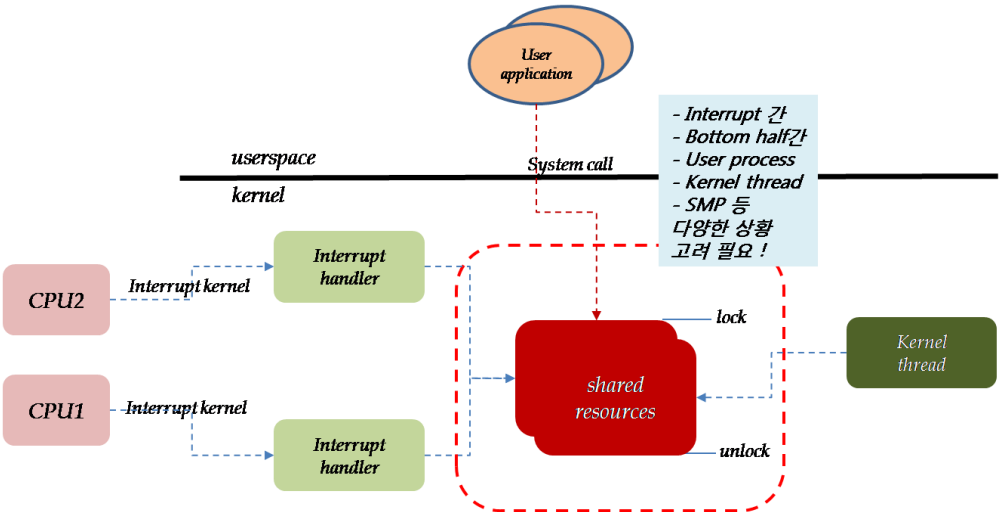


그림 3-1 Concurrency 상황

Kernel programmer가 동기화가 필요한 코드 구현 시 주의할 사항을 정리해 보면 다음과 같다.

<동기화 코드 작성시 주의 사항>

1) Global data인가? 즉, 여기 말고 다른 곳(thread of execution)에서도 이 data에 접근이 가능한가 ?

2) Process context와 interrupt context에서 공유가 가능한 data인가 ?

3) 아니면, 두 개의 서로 다른 interrupt handler에서 공유가 가능한 data인가 ?

4) Data를 사용하던 중에 다른 process에게 CPU를 뺏길 경우, CPU를 선점한 process가 그 data를 access하지는 않는가 ?

5) 현재 process가 sleep하거나 block될 수 있는가 ? 만일 그렇다면, 이때 사용 중이던 data를 어떤 상태로 내버려 두었는가 ?

6) 내가 사용 중이던 data를 해제하려고 하는데, 이를 누군가가 막고 있지는 않은가 (사용할 수 있지는 않은가) ?

7) 이 함수를 시스템의 다른 processor(CPU)에서 다시 호출한다면 어떻게 되는가 ?

8) 내가 짠 code가 concurrency 상황에 안전하다고 확신할 수 있는가 ?

Interrupt handler가 실행 중일 때, 다른 interrupt handler들이 저절로 block되는 것은 아니다(단, 같은 interrupt line은 block을 시킴). 또한, 한 CPU의 interrupt가 disable되었다고, 다른 CPU의 interrupt가 disable되는 것은 아니다. 마지막으로 softirq(tasklett)는 다른 softirq를 선점하지는 않

는다. 이를 표로 정리하면 다음과 같다.

표 3-1 동기화 관련 코드 작성 시 주의 사항

	<Case 1> A) process context B) Tasklet	<Case 2> A) Tasklet B) Tasklet	<Case 3> A) Softirq B) interrupt	<Case 4> A) Interrupt B) Interrupt	<Case 5> A) Work queue B) Kernel thread	<case 6> A) Kernel thread B) Interrupt	비고
A가 B에 의해 선점될 수 있는가 ?	yes	no	yes	yes	yes	yes	local_irq_disable 사용해야함. spin_lock_bh (bottom half 간) mutex (process context 보호 시)
A의 critical section이 다른 CPU에 의해 접근될 수 있는가 ?	yes	yes	yes	yes	yes	yes	spin_lock 사용해야 함.

동기화 기법

리눅스에 자주 사용되는 동기화 기법을 정리해 보면 표 3-2와 같다.

표 3-2 동기화 방법 정리

	Kernel Synchronization Methods	내용 요약/특징
Interrupt context	Atomic operations	
	Spin Locks	Low overhead locking Short lock hold time Need to lock from interrupt context
	Reader-Writer Spin Locks	
process context	Semaphores	
	Reader-Writer Semaphores	
	Mutexes	Long lock hold time Need to sleep while holding lock
	Completion Variables	
	BKL(Big Kernel Lock)	
	Sequential Locks	
	Preemption Disabling	
	Barriers	

Spinlock은 SMP환경에서 사용되도록 만들어 졌으나, 선점형 kernel인 2.6에서는 단일 프로세서 환경에서도 필요하다.

그림 3-2 동기화 방법 정리 – Interrupt Context

Spinlock

1) sleep을 허용하지 않는 코드(interrupt handler)나 sleep하고 싶지 않은 코드(critical sections)를 위해 사용하는 lock 방식이다.

2) 원래는 다중 프로세서 시스템을 위해 만들어 졌다.

3) Spinlock은 잠자지 않으면서, lock이 사용가능해질 때가지 무한 루프를 도는 방식이다(아래 그림 참조)

4) Spinlock으로 보호되는 코드는 kernel preemption이 될 수 없다.

5) 다시 말하지만, spinlock으로 보호되는 critical section 내에서는 절대 sleep해서는 안된다.

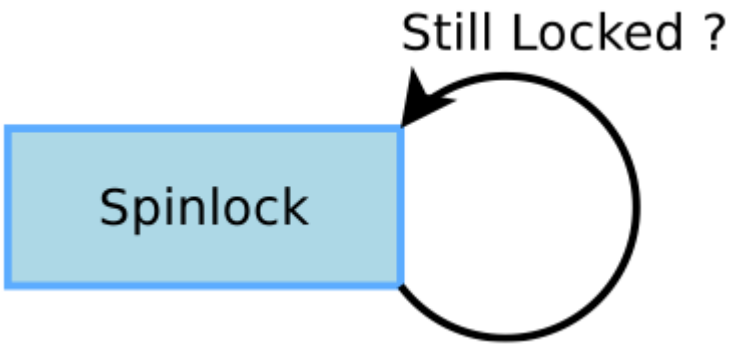


그림 3-3 Spinlock 개념도 [출처 – 참고 문헌 8]

[Spinlock 관련 주요 API 소개 – include/linux/spinlock.h 참조]

<초기화>

DEFINE_SPINLOCK*(my_lock);*
void ***spin_lock_init****(spinlock_t *lock);*

<Lock/Unlock 함수>
void ***spin_lock****(spinlock_t *lock);*

- preempt_disable() 함수 호출 후, spin_acquire() 함수를 호출함.

/ critical sections */*

void ***spin_unlock****(spinlock_t *lock);*

- spin_release() 함수를 호출하고, preempt_enable() 함수 호출함.

void ***spin_lock_irqsave****(spinlock_t *lock, unsigned long flags);*

- Local CPU 상에서 IRQ를 disable 시킨다.

void ***spin_unlock_irqrestore****(spinlock_t *lock, unsigned long flags);*

- Local CPU 상에서 IRQ를 다시 enable 시킨다.

void ***spin_lock_bh****(spinlock_t *lock);*
void ***spin_unlock_bh****(spinlock_t *lock);*

- Hardware interrupt가 아니라, software interrupt(bottom half)를 disable 시켜줌 (local_bh_disable() 함수 사용)
- Bottom half & process context에서 접근 가능할 수 있는 공유 데이터를 보호하기 위해 사용됨.

한 thread가 lock을 얻은 경우, 그 다음에 진입하는 Thread는 wait queue에서 대기(sleep)하게 된다.

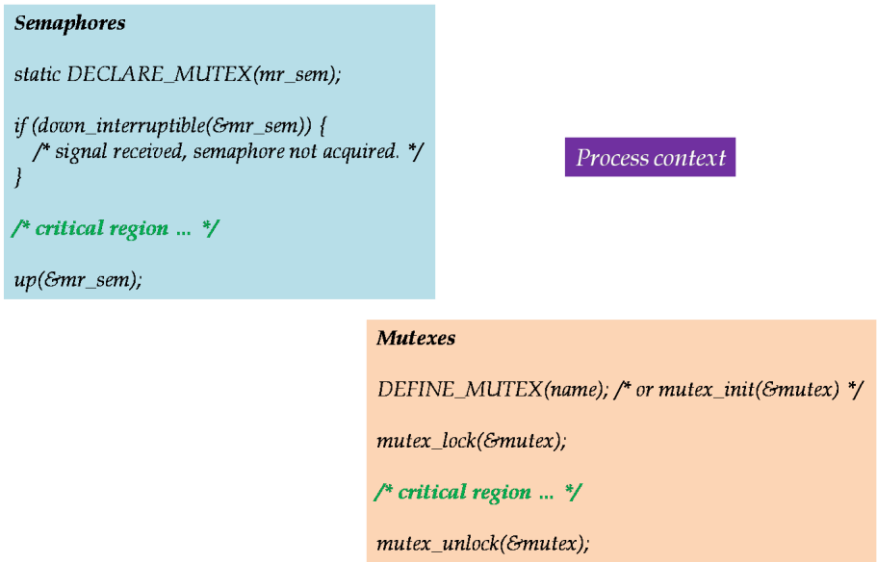


그림 3-4 동기화 방법 정리 – Process Context

Mutex

1) mutex의 커널에서 사용하는 주된 lock 기법이다.

2) 다른 프로세스가 lock을 쥐고 있는 상태에서, 또 다른 프로세스가 lock을 요청하게 되면, 그 프로세스는 block(혹은 sleep)이 되게 된다. 따라서, mutex는 sleep이 허용되는 process context에서 사용 가능하다.

[Mutex 관련 주요 API 소개 – include/linux/mutex.h 참조]

<초기화>

***DEFINE_MUTEX**(name);*

*void **mutex_init**(struct mutex *lock);*

<mutex lock/unlock>

*void **mutex_lock**(struct mutex *lock);*

- *Mutex lock을 얻고자 시도함. 실패할 경우, sleep에 들어감.*
- *Mutex를 가지고 있는 동안에는 프로세스를 종료할 수 없다.*

```
/* critical sections */
```

*void **mutex_unlock**(struct mutex *lock);*

- *지정한 mutex를 해제한다.*

<mutex 관련 기타 API 모음>

*int **mutex_lock_killable**(struct mutex *lock);*

- *mutex_lock과 동일하나, SIGKILL을 써서 mutex를 가지고 있는 프로세스를 강제 종료시킬 수가 있음.*

int ***mutex_lock_interruptible****(struct mutex *lock);*

- 임의의 signal을 사용하여 mutex를 가지고 있는 프로세스를 종료시킬 수 있음.

int ***mutex_trylock****(struct mutex *lock);*

- *지정한 mutex를 얻으려고 시도함. 성공하면 1을 반환, 실패하면 0을 반환함.*

int ***mutex_is_locked****(struct mutex *lock);*

- *lock이 사용 중이면 1을 반환, 그렇지 않으면 0을 반환함.*

Completion

Completion은 세마포어와는 약간 다르게, 서로 다른 두 개의 코드간에 동기(순서 부여)를 맞추고자 할 때 매우 유용하게 사용될 수 있다.

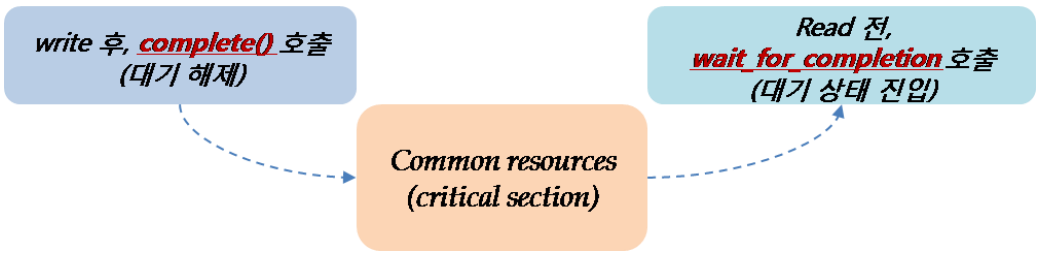


그림 3-5 동기화 방법 정리 – Completion

[Completion 관련 주요 API 소개 – include/linux/completion.h 참조]

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

DECLARE_COMPLETION*(work)*

- *정적으로 completion 초기화함.*

static inline void ***init_completion****(struct completion *x)*

- *동적으로 할당된 completion 을 초기화함.*

void __sched ***wait_for_completion****(struct completion *x)*

- *critical section에 들어갈 때 호출(task가 수행을 완료하기를 기다림)*

int __sched ***wait_for_completion_interruptible****(struct completion *x)*

- *critical section에 들어갈 때 호출(task가 수행을 완료하기를 기다림). 이 함수 호출 동안에 Interrupt 발생 가능함.*

void ***complete****(struct completion *x)*

- *critical section에 들어 갈 수 있도록 해줌(대기 조건을 해지해 줌)*
- *completion을 기다리는 kernel thread에게 이 사실을 알려 줌.*

[예제 코드 - drivers/usb/gadget/omap_udc.c]

```
struct omap_udc {    //See omap_udc.h 파일
        [...]
        struct completion       *done;    //completion 변수 선언    ①
};

static int omap_udc_remove(struct platform_device *pdev)
{
        DECLARE_COMPLETION_ONSTACK(done);
            //정적 completion 초기화. 여기서는 global로 선언하지 않고, 로컬에서 선언함    ②

        if (!udc)
                return -ENODEV;
        usb_del_gadget_udc(&udc->gadget);
        if (udc->driver)
                return -EBUSY;

        udc->done = &done;
        [...]
        wait_for_completion(&done);    //task가 수행을 완료하기를 기다림    ③

        return 0;
}

static void omap_udc_release(struct device *dev)
{
        complete(udc->done);    //대기 조건을 해지해 줌    ④
```

```
        kfree(udc);
        udc = NULL;
}
```

<Preemption Disabling 소개>

아래 preempt_disable( ) 함수를 사용할 경우, kernel preemption을 금지할 수 있다.

```
preempt_disable();
    /* preemption is disabled .. */
preempt_enable();
```

최적화를 위해 compile 시 혹은 CPU에서 실행 시, 상호 dependency가 없는 instruction 들에 대해 순서를 바꾸게 되는데, barrier를 사용하게 되면, 이름 금지시킬 수 있음. 즉, 사용자가 작성한 코드의 순서대로 실행하게 된다. 이 방법은 spinlock 등 locking 기법의 내부를 구성하는 가장 기본적인 방법으로, 앞서 소개한 preempt_disable( ) 함수의 내부도 barrier로 구성되어 있음을 알 수 있다.

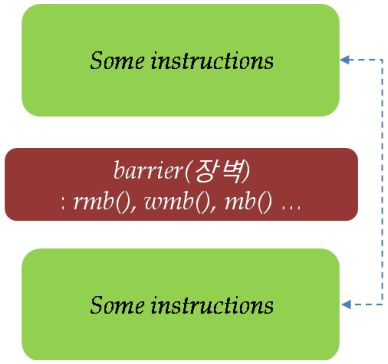


그림 3-6 동기화 방법 정리 – 배리어

<RCU>

<TODO>

2. Notifier Chains

Notifier는 서로 다른 곳에 위치한 커널 코드 간에 event를 전송하기 위한 mechanism으로 callback 함수 개념으로 생각하면 이해가 쉽다. 즉, kernel routine A에서는 호출되기를 원하는 callback 함수를 정의(선언) 및 등록하고, event 발생 시점을 알고 있는 kernel routine B에서는 해당 함수를 호출해 주는 것으로 설명할 수 있을 것이다. 아래 그림 3-7과 3-8을 참조하기 바란다.

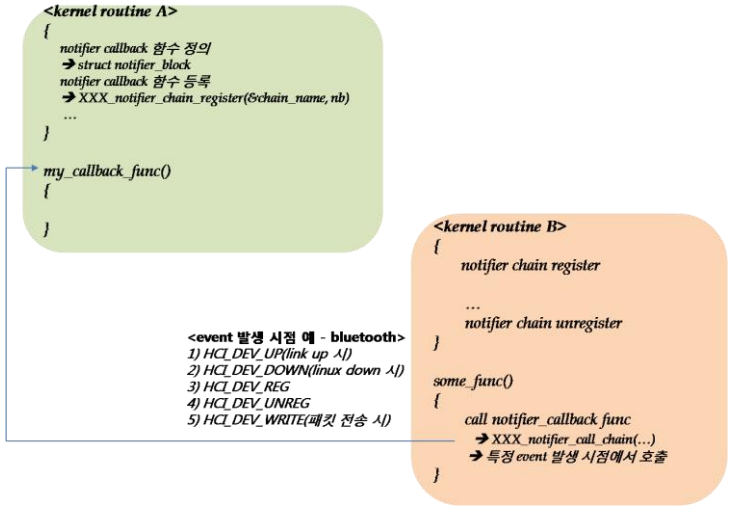


그림 3-7 Notifier 개념(1)

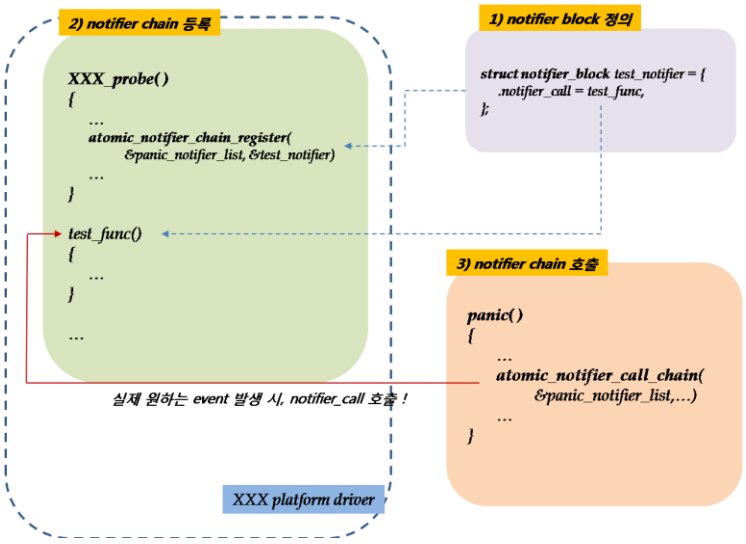


그림 3-8 Notifier 개념(2)

Notifier chain의 종류에는 크게 4가지가 존재하므로, 상황에 따라서 적절히 사용할 수 있어야 한다.

1) Atomic notifier chains

- *Interrupt(atomic) 컨텍스트에서 실행되는 callback chain을 말함. 따라서 blocking될 수 없다(sleep해서는 안됨을 의미함).*

2) Blocking notifier chains

- *1과는 반대로 process 컨텍스트에서 실행되는 callback chain을 말함. 즉, block될 수 있음(수행도중 sleep할 수 있음).*

3) Raw notifier chains

- *Callback 함수, 등록/해제 절차 등에는 아무런 제약이 없는 chain을 말함. 따라서 모든*

lock 관련 사항은 호출함수(caller)에서 책임을 져야 함.

4) SRCU(Sleepable Read-Copy Update) notifier chains

- Blocking notifier chain의 변종으로, rw-semaphore 대신에 SRCU를 사용하는 것이 다름.

[Notifier Chain 관련 주요 API 소개 – include/linux/notifier.h 참조]

```
struct notifier_block {
    notifier_fn_t notifier_call;    //notifier callback function
    struct notifier_block __rcu *next;
    int priority;
};
```

- Notifier block 데이터 구조

```
extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
        struct notifier_block *nb);
extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *nb);
extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *nb);
extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

- 4가지 방식의 notifier chain 등록 함수
- 해제 함수도 있음(X_chain_unregister()). 여기서는 별도로 정리하지 않음.

```
extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v);
extern int __atomic_notifier_call_chain(struct atomic_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v);
extern int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v);
extern int __raw_notifier_call_chain(struct raw_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v);
extern int __srcu_notifier_call_chain(struct srcu_notifier_head *nh,
```

*unsigned long val, void *v, int nr_to_call, int *nr_calls);*

- *4가지 방식의 notifier chain 호출 함수*

[예제 코드 - drivers/hwmon/sht15.c, drivers/regulator/core.c, wm8350-regulator.c]

1) drivers/hwmon/sht15.c - notifier block 선언, callback 함수 정의, notifier chain 등록 주체

```
struct sht15_data {
        [...]
        struct notifier_block          nb;     //notifier block 변수 선언     ①
        [...]
};

static int sht15_invalidate_voltage(struct notifier_block *nb, unsigned long event, void *ignored)
    //callback 함수 정의 – 어디에선가 X_notifier_call_chain() 함수가 호출되면, 실제로 이 함수가 call되는 것임     ⑩
{
        struct sht15_data *data = container_of(nb, struct sht15_data, nb);

        if (event == REGULATOR_EVENT_VOLTAGE_CHANGE)
                data->supply_uv_valid = false;
        schedule_work(&data->update_supply_work);
        return NOTIFY_OK;
}
static int sht15_probe(struct platform_device *pdev)     //probe() 함수     ②
{
        int ret;
        struct sht15_data *data;
        u8 status = 0;

        data = devm_kzalloc(&pdev->dev, sizeof(*data), GFP_KERNEL);
        if (!data)
                return -ENOMEM;
        [...]
        platform_set_drvdata(pdev, data);
        [...]
```

```
        data->nb.notifier_call = &sht15_invalidate_voltage;    //callback 함수 등록    ③
        ret = regulator_register_notifier(data->reg, &data->nb);
            //notifier chain 등록 – regulator framework에서 만든 API    ④
        if (ret) {
                dev_err(&pdev->dev, "regulator notifier request failed₩n");
                regulator_disable(data->reg);
                return ret;
        }
        [...]
}
```

2) drivers/regulator/core.c – Notifier chain API 정의

```
int regulator_register_notifier(struct regulator *regulator, struct notifier_block *nb)
{
    return blocking_notifier_chain_register(&regulator->rdev->notifier, nb);
        //process 컨텍스트에서 실행되는 callback chain을 등록해 줌    ⑤
}
EXPORT_SYMBOL_GPL(regulator_register_notifier);

int regulator_unregister_notifier(struct regulator *regulator, struct notifier_block *nb)
{
    return blocking_notifier_chain_unregister(&regulator->rdev->notifier, nb);
        //process 컨텍스트에서 실행되는 callback chain을 해제해 줌.
}
EXPORT_SYMBOL_GPL(regulator_unregister_notifier);

static void _notifier_call_chain(struct regulator_dev *rdev, unsigned long event, void *data)
{
        /* call rdev chain first */
        blocking_notifier_call_chain(&rdev->notifier, event, data);
            //notifier chain을 호출해 줌 – 이 함수가 호출되어야, 비로소 최초 등록한 callback 함수가 불리어지게 됨    ⑨
}

int regulator_notifier_call_chain(struct regulator_dev *rdev, unsigned long event, void *data)    ⑦
{
```

```
        _notifier_call_chain(rdev, event, data);      ⑧
        return NOTIFY_DONE;
}
EXPORT_SYMBOL_GPL(regulator_notifier_call_chain);
```

3) regulator/wm8350-regulator.c – notifier chain 호출, 즉 event 발생

```
    //위의 core.c 파일에서 제공하는 API를 이용하여,
    //sht15.c의 callback 함수(sht15_invalidate_voltage) 호출.
    //참고로, core.c에서도 중간 중간에 notifier chain을 호출하는 곳이 보임.
static irqreturn_t pmic_uv_handler(int irq, void *data)
{
        struct regulator_dev *rdev = (struct regulator_dev *)data;
        struct wm8350 *wm8350 = rdev_get_drvdata(rdev);

        mutex_lock(&rdev->mutex);
        if (irq == WM8350_IRQ_CS1 || irq == WM8350_IRQ_CS2)
                regulator_notifier_call_chain(rdev, REGULATOR_EVENT_REGULATION_OUT, wm8350);
                  // 궁극적으로 blocking_notifier_call_chain() 함수를 호출하게 됨      ⑥
        else
                regulator_notifier_call_chain(rdev, REGULATOR_EVENT_UNDER_VOLTAGE, wm8350);
        mutex_unlock(&rdev->mutex);

        return IRQ_HANDLED;
}
```

3. 메모리 관리 기법

이번 절에서는 아래의 관점에서 메모리를 바라보고 내용을 정리해 보고자 한다.

1) virtual address 개념 및 kernel memory map

- *physical memory가 (현실적으로) 작으므로, virtual address가 어떠한 방식으로 운용되는지 파악해야 함.*
- *Kernel memory map도 더불어 이해하고자 함.*

2) memory allocation schemes …

- *kernel에서 memory를 어떠한 형태로 할당해 주고 있는지 파악해야 함.*
- *Page(Buddy), slab, slub, slob allocator …*
- *Kmalloc, vmalloc …*

3) kernel과 user space간의 데이터 전달(공유) 방법

4) 안드로이드 커널의 메모리 관련 기능

- *ION*
- *Ashmem*
- *Low Memory Killer*

3.1 Virtual memory 개념과 Kernel Memory Map

임베디드 시스템의 경우, 일반적으로 physical memory가 작으므로 이를 해결하기 위하여, MMU를 활용하여 virtual memory 방식으로 운용된다. 32bit CPU의 경우 2^{32} = 4GB의 가상 주소를 사용할 수 있다. 각각의 process는 자신만의 4GB virtual address space를 사용할 수 있는데, /proc/<pid>/maps 내용을 보면, 서로 다른 process가 동일한 위치(주소)를 사용하고 있음을 알 수 있다.

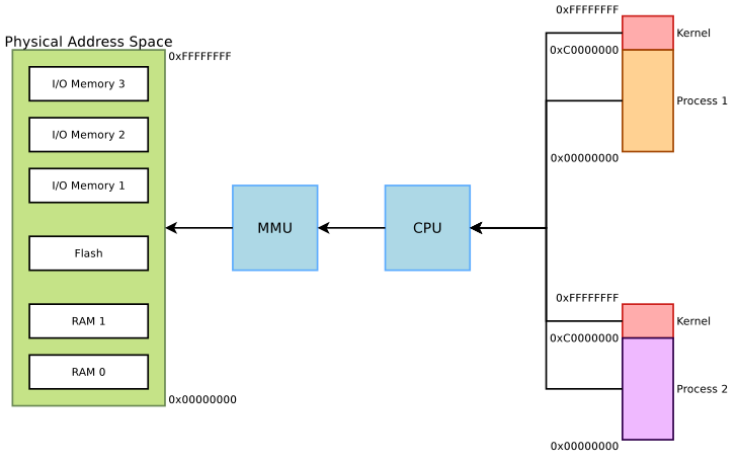


그림 3-9 Physical & Virtual 메모리 관계 [출처 – 참고 문헌 8]

커널은 물리적인 페이지(page)를 기본 단위로하여 메모리를 관리하며, 비슷한 특성을 가진 페이지를 모아 메모리 존(zone)을 구분해 두고 있다.

1) ZONE_DMA

- 이 영역(zone)은 DMA를 수행 가능한 페이지를 포함하고 있음.
- 아래 ZONE_NORMAL 영역에서 DMA를 수행할 수 없을 경우에 사용되며, ARM의 경우 크기가 가변적이다.

2) ZONE_NORMAL

- 이 영역은 정상적으로 mapping이 가능한 페이지를 포함하고 있음.
- 대부분 이 영역이라고 봐도 무방함.

3) ZONE_HIGHMEM

- 이 영역은 커널 주소 공간에 영구적으로 매핑되지 못하는 high memory를 포함하고 있음.

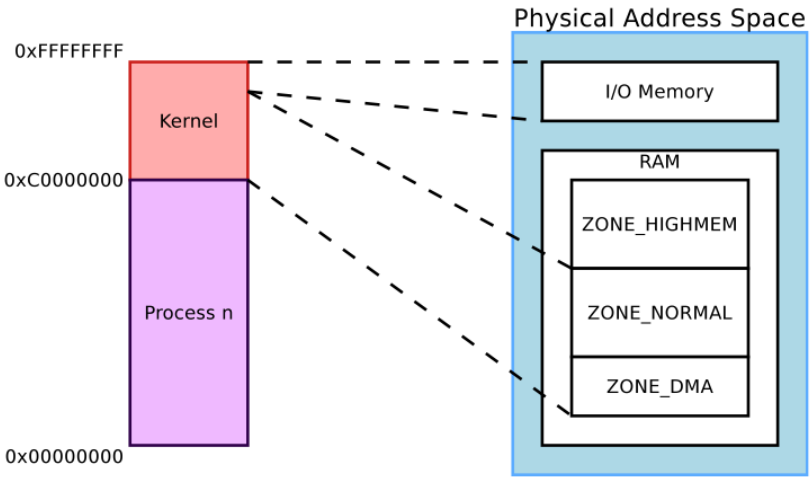


그림 3-10 Physical/Virtual 영역 매핑 - 커널 영역 [출처 – 참고 문헌 8]

아래 User space map 정보는 prelink-linux-arm.map을 참조하여 작성한 것일 뿐, 실제 동작중인 내용(주소 값)은 다를 수 있다(단, 각 영역의 순서/위치는 일치함).

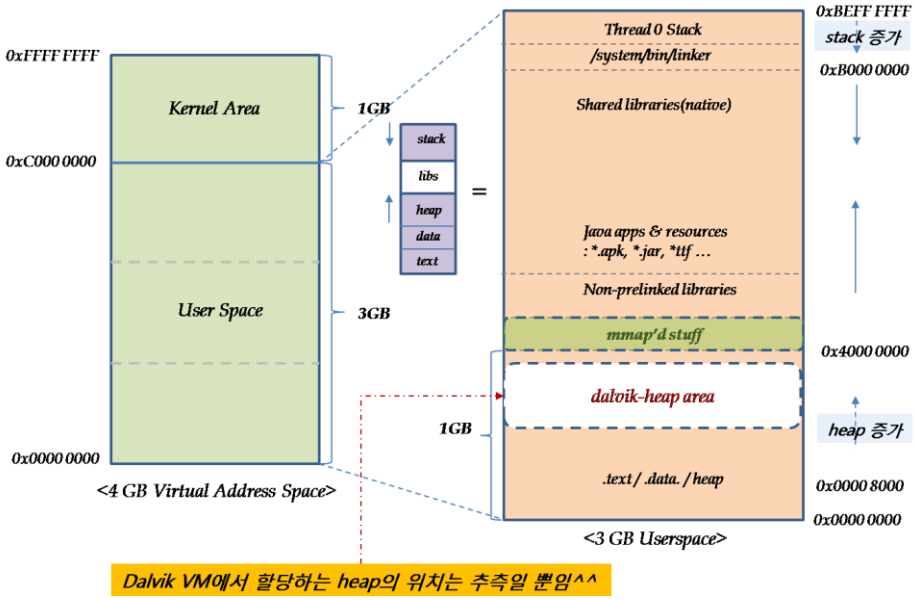


그림 3-11 Physical/Virtual 영역 매핑 - 사용자 영역

3.2 메모리 할당 기법

커널에서 메모리를 할당하는 기법은 아래와 같이 정리할 수 있으며, 그림 3-12에 하나로 표현하였다.

1) Page 할당자

- 물리적으로 연속된 페이지(4K, 8K, 16K, 32K ...) 단위의 메모리 할당

2) Slab, slub, slob 할당자

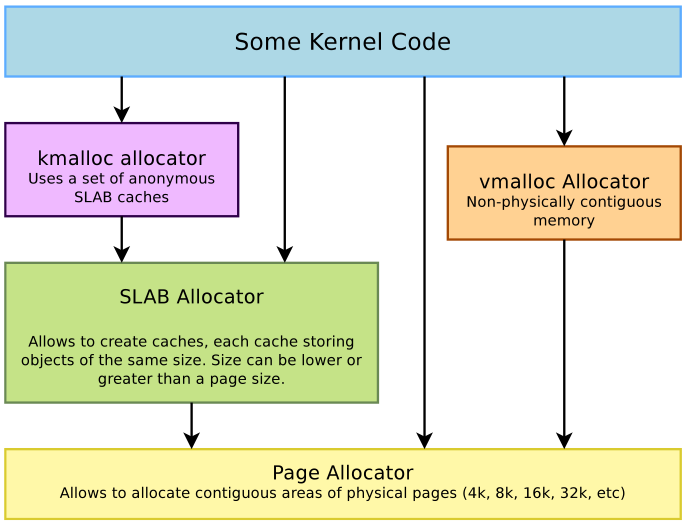
- 캐쉬 영역을 만들고 관리해 줌(속도를 빠르게 할 목적).

3) Kmalloc 할당자

- 물리적으로 연속된 메모리 영역을 할당해 줌. 내부적으로 Slab 캐쉬 영역을 사용함.

4) Vmalloc 할당자

- 임의의 커널 메모리 영역을 할당해 줌. 물리적으로 연속적이지 않음. Slab 캐쉬와는 무관함.



그린 3-12 커널 메모리 할당자(allocator) [출처 – 참고 문헌 8]

Page 할당자

Page 할당자의 의미와 그 특징을 정리하면 다음과 같다.

1) 중간 크기의 메모리 할당에 적합한 방식이다.

2) 한 페이지는 보통 4K를 말하지만, 아키텍처에 따라서 더 클 수도 있다(가령, sh나 mips에서는 4, 8, 16 혹은 64 KB까지 가능하지만, x86이나 arm에서는 불가하다).

3) 버디(buddy) 할당 방식은 2^n 크기의 페이지만을 할당하는 방식이다.

4) 일반적으로 페이지의 최대 크기는 8192KB이지만, 이것도 커널 설정에 따라 달라질 수 있다.

5) 페이지 할당자에 의해서 할당된 메모리 영역은 물리적으로 연속적이다.

6) 이 말은 조각된 물리 메모리 때문에, 커다란 메모리 영역을 확보하는 것이 어려움을 의미한다.

[페이지 할당 관련 주요 API 소개 – include/linux/gfp.h 참조]

alloc_pages*(gfp_mask, order)*

- 매크로임
- 2^{order} 개의 연속된 물리 페이지를 할당하고, 시작 페이지의 page structure에 대한 주소(pointer)를 돌려 줌.

unsigned long ***get_zeroed_page****(gfp_t gfp_mask)*

- 페이지를 할당해 준 후, 0으로 초기화함.

unsigned long ***__get_free_page****(gfp_t gfp_mask)*

- 페이지를 할당해 주나, 초기화하지 않음.

unsigned long ***__get_free_pages****(gfp_t gfp_mask, unsigned int order)*

- 페이지를 할당해 줌.
- 2^{order} 개의 연속된 물리 페이지를 할당하고, 시작 페이지의 논리 주소에의 포인터를 돌려줌.

void ***free_page****(unsigned long addr)*

- 할당했던 페이지 하나를 해제함.

void ***free_pages****(unsigned long addr, unsigned int order)*

- 할당했던 여러 페이지를 해제해 줌. 할당시의 순서와 동일해야 할 필요 있음.

<메모리 할당 플래그>

GFP_KERNEL

- 일반적인 커널 메모리 할당 플래그로, 충분한 메모리 영역을 확보하기 위해 block(sleep)될 수도 있다. Interrupt 컨텍스트의 경우에는 사용이 불가하다.

GFP_ATOMIC

- Interrupt 컨텍스트(혹은 critical section 내)에서 메모리 할당 시 사용하는 플래그로, 메모리 할당 중 block되지 않는다. 따라서 가용 메모리가 존재하지 않을 경우, 할당에 실패할 수도 있는 방식이다.

GFP_DMA

- DMA 전송이 가능한 물리 메모리 영역에서 메모리를 할당 받고 싶을 때 사용하는 플래그이다.

이 밖에도 include/linux/gfp.h 파일을 보면, 사용 가능한 플래그가 더 정의되어 있으니, 확인해 보기 바란다.

Slab 할당자

Slab 할당자의 의미와 그 특징을 정리하면 다음과 같다.

1) SLAB 할당자는 캐쉬를 할당하도록 해준다.

2) 캐쉬를 구성하는 객체의 크기는 page 크기보다 작거나 클 수도 있다.

3) SLAB 할당자는 할당된 객체의 개수에 따라서 필요 시 캐쉬의 크기를 늘리거나 줄이기도 한다.

SLAB 할당자가 페이지를 할당하거나 해제하는 작업은 실제로 page 할당자에서 해준다.
4) SLAB 캐쉬는 directory entries, file objects, network packet descriptors, process descriptors 등 많은 경우에 사용되고 있다.

[페이지 할당 관련 주요 API 소개 – include/linux/slab.h 참조]

*void __init **kmem_cache_init**(void);*

*struct kmem_cache ***kmem_cache_create**(const char *, size_t, size_t, unsigned long, void (*)(void *));*

*void **kmem_cache_destroy**(struct kmem_cache *);*

*int **kmem_cache_shrink**(struct kmem_cache *);*

*void **kmem_cache_free**(struct kmem_cache *, void *);*

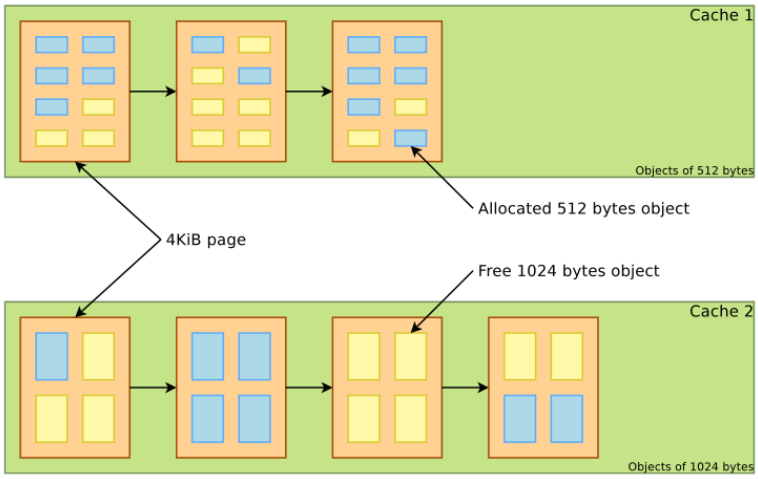


그림 3-13 Slab 할당자 [출처 – 참고 문헌 8]

SLAB 할당자에는 크게 3가지 종류(변종)가 있다.

SLAB 할당자

- 초기 방식(Linux 2.6에서 잘 동작함)

SLOB 할당자

- 메모리가 작은 시스템에 적합하도록 만들어진 방식으로 SLAB에 비해 훨씬 단순한 구조임.

SLUB 할당자

- 2.6.23 버전 이후로 새롭게 표준이 된 방식. SLAB보다 단순하며, 대용량 메모리에서 보다 효과적인 방식임. 안드로이드 스마트폰의 경우는 메모리가 대개 1GB 이상이므로 이

방식을 채용하고 있음.

Kmalloc 할당자

kmalloc 할당자의 의미와 그 특징을 정리하면 다음과 같다.

1) Kmalloc 할당자는 가장 기본적인 커널 메모리 할당자이다.

2) 작은 크기의 메모리를 할당 받기 위해서는 /proc/slabinfo에 있는 kmalloc-XXX 라는 이름의 SLAB 캐쉬를 활용한다.

3) 큰 크기의 메모리를 할당 받기 위해서는 page 할당자를 직접 이용한다.

4) 할당 받은 메모리 영역은 물리적으로 연속적이어야 한다.

5) Page 할당자가 사용하는 플래그(GFP_KERNEL, GFP_ATOMIC, GFP_DMA, 등)와 동일한 플래그를 사용한다.

6) X86이나 ARM에서 할당 가능한 크기는 매 할당 시 마다 4MB이며, 전체 합이 128MB이다.

7) 커널 프로그래밍 시, 다른 메모리 할당 방식을 사용해야만 하는 명백한 이유가 없다면, 무조건 이 할당자를 사용해야 한다(기본 할당자임).

[kmalloc 할당 관련 주요 API 소개 – include/linux/slab.h 참조]

*void **__kmalloc__*(size_t size, int flags);*

- Size 바이트의 영역을 할당하고, 그 시작 주소(가상 주소임)를 돌려준다.
- size: 할당 받을 바이트의 수
- flags: same ags as the page allocator
- flags: page 할당자에서 사용했던 플래그와 동일함.

*void __kfree__(const void *objp);*

- 할당 받았던 영역을 해제한다.

*void *__kzalloc__(size_t size, gfp_t flags);*

- Kmalloc과 동일하나, 할당 후, 버퍼의 내용을 0으로 초기화한다.

*void *__kcalloc__(size_t n, size_t size, gfp_t flags);*

- n(배열의 개수) x size 크기의 배열에 해당하는 메모리 영역을 확보하고, 내용을 0으로 초기화시킨다.

*void *__krealloc__(const void *p, size_t new_size, gfp_t flags);*

- p가 가리키는 버퍼의 크기를 새로운 크기인 new_size로 조정한다. 이때, new_size가 기존의 크기보다 크다면, 버퍼를 새로 할당하고, data 내용도 복사해준다.

Vmalloc 할당자

vmalloc 할당자의 의미와 그 특징을 정리하면 다음과 같다.

1) Vmalloc 할당자는 물리적으로 연속되지 않지만, 가상 메모리 상으로는 연속된 영역을 할당받고자 할 때 사용하는 방식이다.
2) Vmalloc에 의해 할당된 메모리 영역은 커널 주소 공간은 맞으나, 1:1 매핑 이외의 영역이다.
3) vmalloc으로 할당 받을 수 있는 영역은 거의 가용 메모리 전체의 크기에 해당할 만큼 매우 크다. 이는 물리 메모리 단편화(fragmentation)가 더 이상 문제가 되지 않기 때문이다. 참고로, vmalloc에 의해 할당된 영역은 DMA를 위한 용도로는 사용이 불가하다. 그 이유는 DMA의 경우는 물리적으로 연속된 버퍼를 요구하기 때문이다.

[kmalloc 할당 관련 주요 API 소개 – include/linux/vmalloc.h 참조]

*void **vmalloc**(unsigned long size);*

- Size 바이트 만큼의 메모리를 할당하고, 가상 주소를 돌려 준다.

*void **vfree**(void *addr);*

- 할당 받았던 메모리를 해제한다.

3.3 사용자 영역과 커널 영역간의 데이터 교환 방법

put(get)_user() 함수와 copy_to(from)_user() 함수

사용자 영역의 응용 프로그램과 커널 영역은 서로 다른 가상 주소 공간을 사용한다. 따라서 응용 프로그램에서 사용하던 주소를 커널에 전달(가령, read(), write() 혹은 ioctl() 함수 사용)하게 되면, 그 주소는 커널에서 직접적으로 사용할 수가 없게 된다. 응용 프로그램에서 전달받은 주소를 사용하는 커널 코드에서는 일반적으로 아래의 2가지 절차를 밟아야 한다.

1) 사용자 영역의 주소를 검사하여, 해당 page가 메모리 상에 없을 경우, page fault가 발생할 수 있으므로 이를 처리해 주어야 한다.
2) 사용자 영역과 커널 영역간의 데이터 복사 절차를 수행한다.

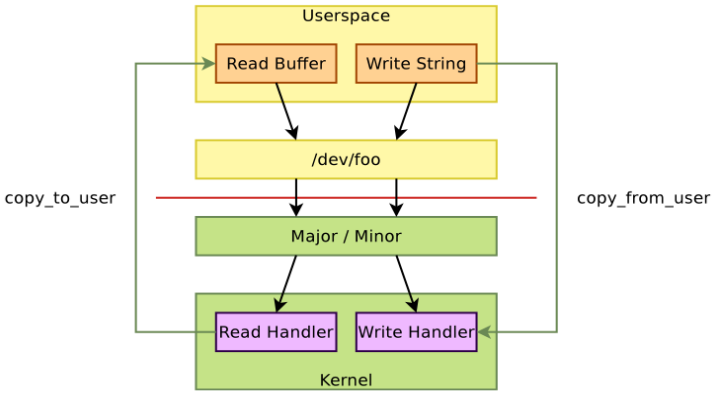


그림 3-14 copy_from(to)_user 함수의 이해(1)

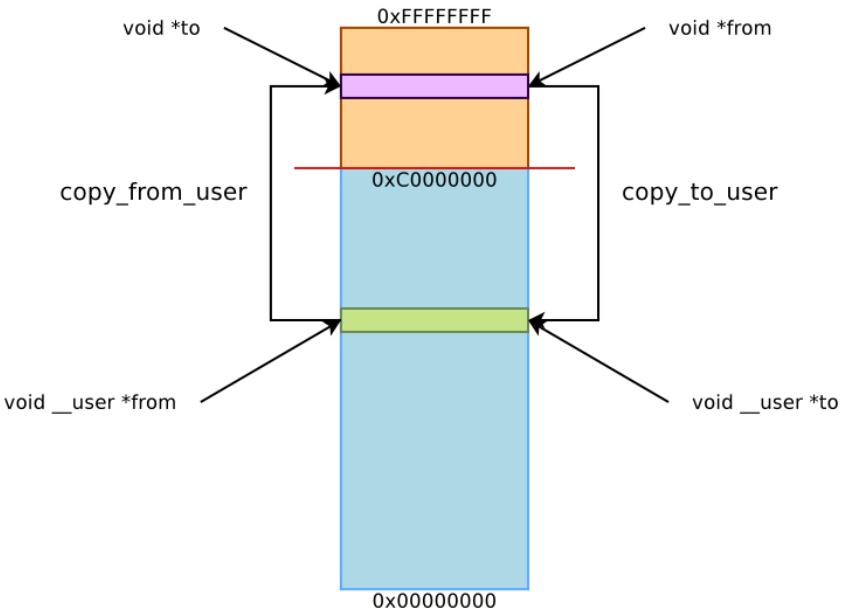


그림 3-15 copy_from(to)_user 함수의 이해(2)

[주요 API 소개 – arch/arm/include/asm/uaccess.h 참조]

get_user*(x,p)*

- 매크로임. User space로부터 kernel space로 데이터 전달(복사)
- 포인터 p(사용자 영역)가 가리키는 데이터를 x(커널 영역)에 대입(x <- p)

put_user*(x,p)*

- 매크로임. Kernel space로부터 user space로 데이터 전달(복사)
- 포인터 p(사용자 영역)가 가리키는 위치에 x (커널 영역) 데이터를 대입(x -> p)

static inline unsigned long __must_check ***copy_from_user****(void *to, const void __user *from, unsigned long n)*

- user space(from)로부터 kernel space(to)로 n bytes를 복사함.

static inline unsigned long __must_check **copy_to_user**(void __user *to, const void *from, unsigned long n)

- kernel space(from)로부터 user space(to)로 n bytes를 복사함.

Mapping User Pages

앞서 제시한 방법 과는 달리, 사용자 영역의 메모리를 커널에서 직접 사용하는 방법(복사 과정이

필요 없음)에 관하여 정리해 보고자 한다.

[주요 API 소개 – include/linux/mm.h 참조]

*long **get_user_pages**(struct task_struct *tsk, struct mm_struct *mm,*
unsigned long start, unsigned long nr_pages,
*int write, int force, struct page **pages,*
*struct vm_area_struct **vmas);*

- user space의 메모리를 kernel에서 직접 사용할 수 있도록 해줌.
- tsk: user process(=> current), mm: current->mm
- start: 사용자 영역 버퍼의 시작 주소, len: page 길이
- pages: page structure를 가리키는 포인터 배열. 이 값을 가지고, kernel address 추출하게 됨.

[예제 코드 - drivers/video/pvr2fb.c]

```
static ssize_t pvr2fb_write(struct fb_info *info, const char *buf, size_t count, loff_t *ppos)
  // 사용자 영역의 응용 프로그램에서 write() 함수 호출     ①
{
        unsigned long dst, start, end, len;
        unsigned int nr_pages;
        struct page **pages;
        int ret, i;

        nr_pages = (count + PAGE_SIZE - 1) >> PAGE_SHIFT;
        pages = kmalloc(nr_pages * sizeof(struct page *), GFP_KERNEL);
        if (!pages)
                return -ENOMEM;

        down_read(&current->mm->mmap_sem);
        ret = get_user_pages(current, current->mm, (unsigned long)buf,
                                nr_pages, WRITE, 0, pages, NULL);
                // 사용자 영역의 buf로부터, kernel page pointer 추출     ②
        up_read(&current->mm->mmap_sem);
        if (ret < nr_pages) {
                nr_pages = ret;
                ret = -EINVAL;
                goto out_unmap;
        }
```

```
        dma_configure_channel(shdma, 0x12c1);

        dst     = (unsigned long)fb_info->screen_base + *ppos;
        start = (unsigned long)page_address(pages[0]);
              //page에 해당하는 가상 주소 전환     ③
        end     = (unsigned long)page_address(pages[nr_pages]);
        len     = nr_pages << PAGE_SHIFT;

        /* Half-assed contig check */
        if (start + len == end) {
                /* As we do this in one shot, it's either all or nothing.. */
                if ((*ppos + len) > fb_info->fix.smem_len) {
                        ret = -ENOSPC;
                        goto out_unmap;
                }

                dma_write(shdma, start, 0, len);     //커널 가상 주소에 값 쓰기     ④
                dma_write(pvr2dma, 0, dst, len);
                dma_wait_for_completion(pvr2dma);

                goto out;
        }
        [...]

out_unmap:
        for (i = 0; i < nr_pages; i++)
                page_cache_release(pages[i]);     // 다 사용한 page를 해제     ⑤
        kfree(pages);

        return ret;
}
```

Memory Mapping

mmap을 이용할 경우, application process에서 메모리 복사 과정 없이, 직접 kernel 공간을 사용할 수 있다(get_user_pages()와 반대 개념). 반면, read/write/ioctl의 경우는 process memory와

kernel memory 사이에 메모리 copy 과정이 수반된다. mmap의 개념을 그림으로 표현해 보면 다음과 같다.

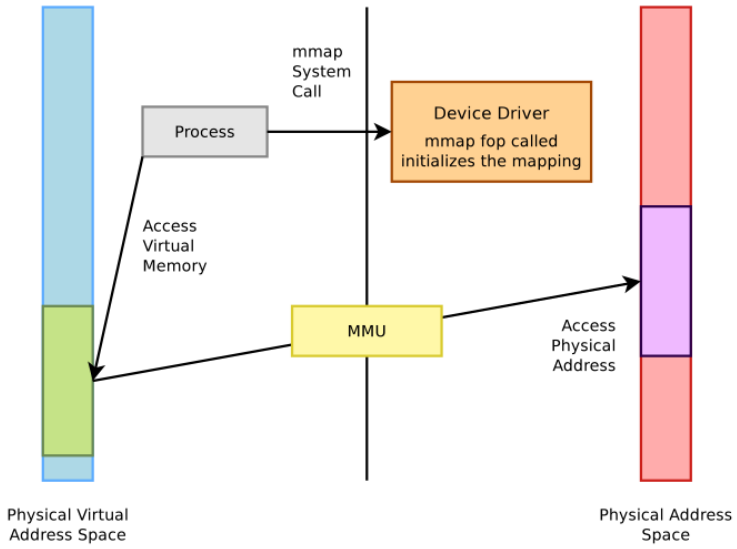


그림 3-16 mmap의 이해(1) [출처 – 참고 문헌 8]

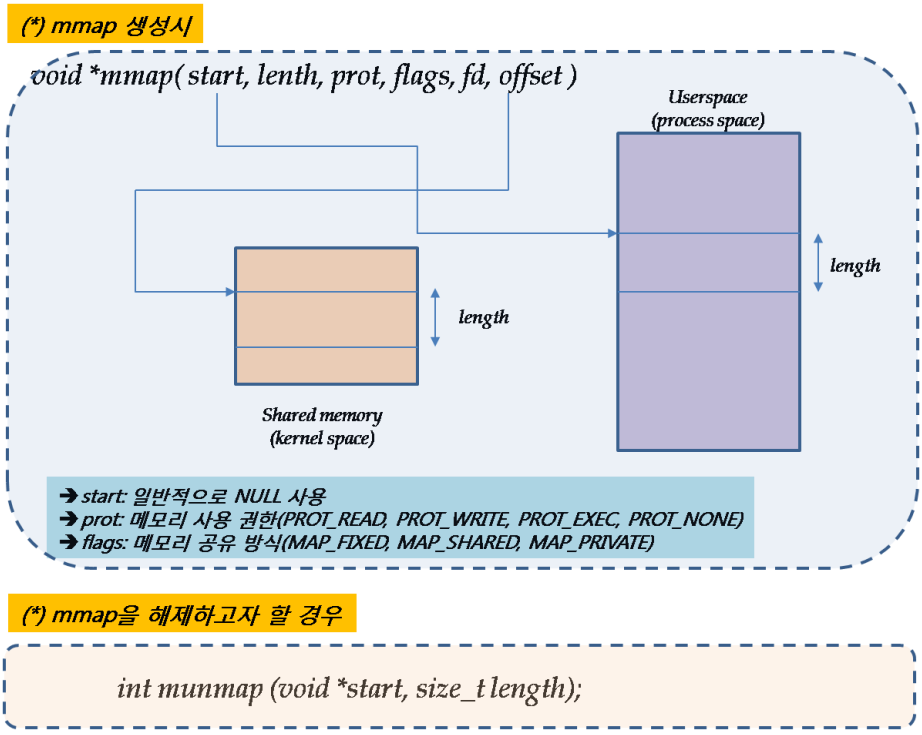


그림 3-17 mmap의 이해(2)

응용 프로그램에서 mmap() 함수를 호출할 경우, kernel 내에서 어떠한 함수들이 호출되는지를 살펴 봄으로써 mmap의 동작 원리를 파악해 볼 수 있을 것이다.

1) sys_mmap2 호출(arch/arm/kernel/call.S)

```
CALL(sys_mmap2)
```

2) sys_mmap_pgoff() 호출(arch/arm/kernel/entry-common.S)

```
#if PAGE_SHIFT > 12
        tst r5, #PGOFF_MASK
        moveq     r5, r5, lsr #PAGE_SHIFT – 12
        streq     r5, [sp, #4]
        beq sys_mmap_pgoff
        mov r0, #-EINVAL
        mov pc, lr
#else
        str r5, [sp, #4]
        b     sys_mmap_pgoff
#endif
ENDPROC(sys_mmap2)
```

3) vm_mmap_pgoff() 호출(mm/mmap.c)

```
SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,
            unsigned long, prot, unsigned long, flags, unsigned long, fd, unsigned long, pgoff)
{
    [...]
    retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
    if (file)
        fput(file);
    [...]
}
```

4) do_mmap_pgoff() 호출(mm/util.c)

```
unsigned long vm_mmap_pgoff(struct file *file, unsigned long addr,
                unsigned long len, unsigned long prot, unsigned long flag, unsigned long pgoff)
{
    [...]
    ret = do_mmap_pgoff(file, addr, len, prot, flag, pgoff, &populate);
    [...]
}
```

5) mmap_region() 함수 호출(mm/mmap.c)

```
unsigned long do_mmap_pgoff(struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot, unsigned long flags, unsigned long pgoff,
        unsigned long *populate)
{
        [...]
        addr = get_unmapped_area(file, addr, len, pgoff, flags);
        [...]
        addr = mmap_region(file, addr, len, vm_flags, pgoff);  //커널 메모리 할당 후, 매핑 시도
        [...]
        return addr;
}
```

6) 아래 함수 안에서 커널 영역을 할당하고, 이를 프로세스 영역에 매핑시켜준다.

```
unsigned long mmap_region(struct file *file, unsigned long addr,
          unsigned long len, vm_flags_t vm_flags, unsigned long pgoff)
{
    [...]
    vma = vma_merge(mm, prev, addr, addr + len, vm_flags, NULL, file, pgoff, NULL);
       //전후의 프로세스 공간 영역과의 연결 시도(vm_area_struct 조작)
    [...]
    vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
      //vm_area_struct 영역 할당. 이 영역이 사용자 공간(프로세스 공간)에 매핑되는 영역임
    [...]
    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags;
    vma->vm_page_prot = vm_get_page_prot(vm_flags);
    vma->vm_pgoff = pgoff;
    INIT_LIST_HEAD(&vma->anon_vma_chain);
    [...]
    if (file) {   //struct file에 대한 처리를 진행함.
        [...]
        vma->vm_file = get_file(file);
        error = file->f_op->mmap(file, vma);
            //사용자 영역에서 open한 장치 드라이버의(*mmap) callback 호출
        [...]
        addr = vma->vm_start;     //위의 mmap() callback 후 변경 사항 반영
        pgoff = vma->vm_pgoff;
```

```
        vm_flags = vma->vm_flags;
    }
    [...]
    vma_link(mm, vma, prev, rb_link, rb_parent);
        //위에서 생성한 vm_area_struct를 vm_list & rbtree에 연결
    file = vma->vm_file;
    return addr;    //매핑된 주소를 반환
    [...]
}
```

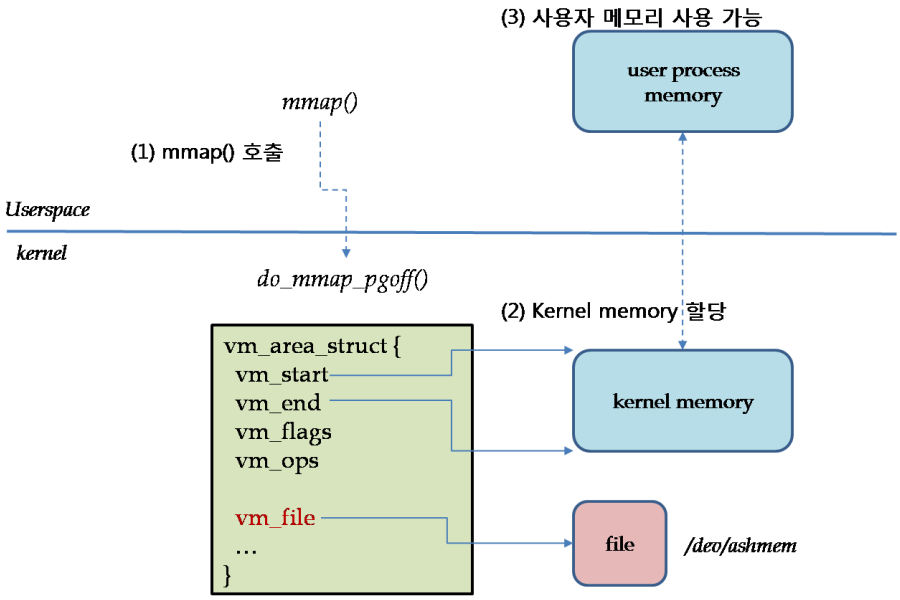


그림 3-18 mmap의 이해(3)

지금까지 설명한 mmap의 개념은 이후 설명할 Ashmem, ION 메모리 등 다양한 영역에서 널리 사용되고 있다.

3.4 안드로이드 메모리 관련 기법

Ashmem(Anonymous Shared Memory) 소개

Ashmem은 (named) shared memory와 유사한 기법으로, name이 아닌 file을 통해 서로 다른 프로세서에서 상호 접근이 가능하도록 만든 공유 메모리 기법이다(참고문헌 [1]의 저자이기도 한, Google의 Robert Love가 만듦). Ashmem을 위해서는 앞서 이미 설명한 mmap 개념이 들어가게 되며, file descriptor 전달을 위해서는 binder 메커니즘을 이용해야 한다.

사용자 영역의 응용 프로그램에서 ashmem을 사용하기 위한 절차를 간략히 요약해 보면 다음과

같다.

<ashmem 생성 프로세스>

1) fd = open /dev/ashmem

2) ioctl(fd, ASHMEM_SET_NAME, buf);

- ashmem 이름 지정(여러 개의 ashmem 생성시 구분 목적)
- 1, 2)의 내용을 줄여서 아래 함수 호출
  - fd = **ashmem_create_region**("my_shm_region", size);

3) mmap(...fd...) 함수 호출

<ashmem 사용 프로세스>

1) binder를 통해 fd를 얻어옴

2) fd2 = dup(fd);

3) mmap(...fd2...) 함수 호출

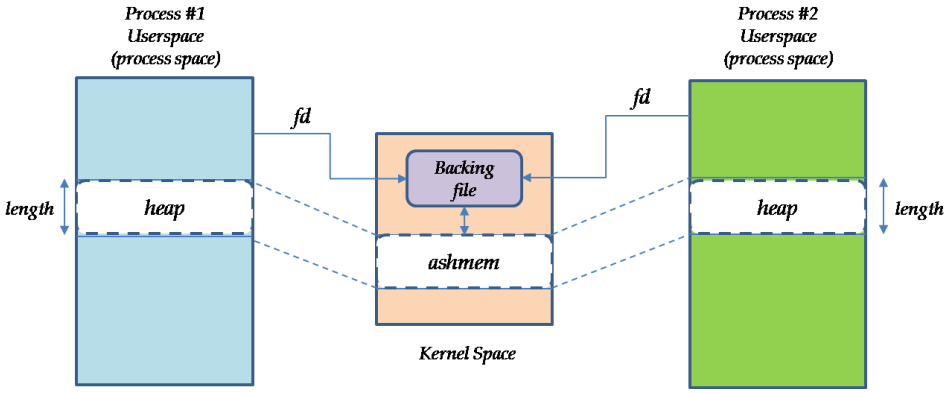


그림 3-19 Ashmem 소개(1)

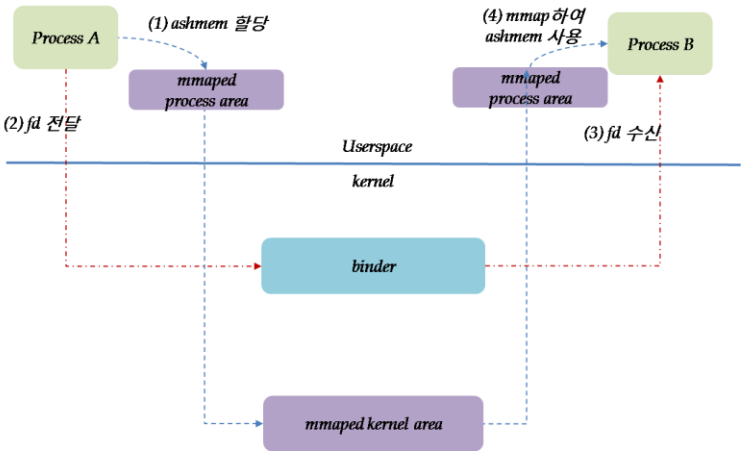


그림 3-20 Ashmem 소개(2)

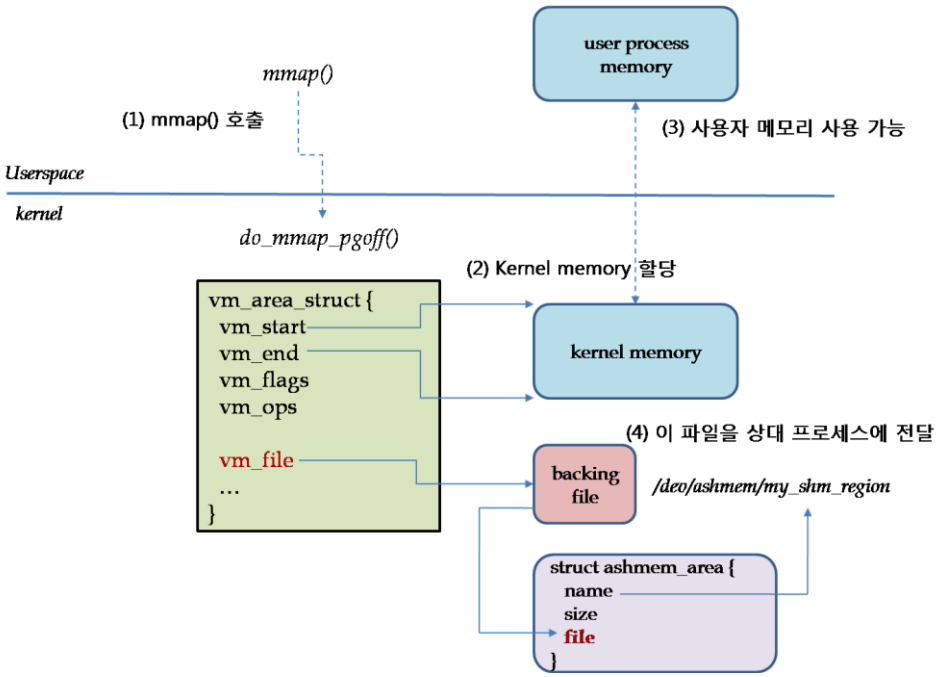


그림 3-21 Ashmem 소개(3)

Ashmem 생성 절차를 정리해 보면 다음과 같다.

1) /dev/ashmem 파일을 오픈한다.

- ashmem_area 공간을 확보한 후 내용을 추가 한다. 이후 file->private_data에 이 정보를 지정하는 일을 한다.

2) mmap()을 통해 사용자 영역의 버퍼에 매핑되는 커널 메모리(vm_area_struct 사용)를 할당하고, 매핑한다.

3) mmap() system call을 수행하는 중에 ashmem_open() 함수를 호출하게 되고, 아래의 작업을 수행한다.

- 프로세스간에 공유할 메모리는 이미 mmap() system call을 통해 할당 받은 상태임.
- 따라서, 이 함수에는 프로세스간의 공유를 위한 메모리 영역을 할당하는 것이 아니라, 공유를 위한 매개체인 file을 하나 만들고, 이 것에 이름을 새로 부여하는 것이 목적임.

4) ashmem_open() 함수에서 복귀한 후, 변경 사항을 반영한다.

- vm_area_struct의 vm_file이 backing 파일로 변경됨(ashmem_mmap()에서 생성한 파일)

5) 이후, 새롭게 생성된 backing 파일을 상대방 프로세스에 전달하여 kernel memory를 서로 공유하게 된다.

[예제 코드 - drivers/staging/android/ashmem.c]

```
struct ashmem_area {
```

```
        char name[ASHMEM_FULL_NAME_LEN]; /* optional name in /proc/pid/maps */
        struct list_head unpinned_list;   /* list of all ashmem areas */
        struct file *file;          /* the shmem-based backing file */
        size_t size;                /* size of the mapping, in bytes */
        unsigned long prot_mask;    /* allowed prot bits, as vm_flags */
};

struct ashmem_range {
        struct list_head lru;          /* entry in LRU list */
        struct list_head unpinned;   /* entry in its area's unpinned list */
        struct ashmem_area *asma;    /* associated area */
        size_t pgstart;              /* starting page, inclusive */
        size_t pgend;                /* ending page, inclusive */
        unsigned int purged;         /* ASHMEM_NOT or ASHMEM_WAS_PURGED */
};

static int ashmem_open(struct inode *inode, struct file *file)
    // 이 함수에서는 ashmem_area 공간을 확보한 후 내용을 추가 한다.
    // 이후 file->private_data에 이 정보를 지정하는 일을 함.
{
        struct ashmem_area *asma;
        int ret;

        ret = generic_file_open(inode, file);
        if (unlikely(ret))
                return ret;

        asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
        if (unlikely(!asma))
                return -ENOMEM;

        INIT_LIST_HEAD(&asma->unpinned_list);
        memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
        asma->prot_mask = PROT_MASK;
        file->private_data = asma;

        return 0;
}
```

```
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
  //프로세스간에 공유할 메모리는 이미 mmap() system call을 통해 할당 받은 상태임.
  //따라서, 이 함수에는 프로세스간의 공유를 위한 메모리 영역을 할당하는 것이 아니라,
  //공유를 위한 매개체인 file을 하나 만들고, 이 것에 이름을 새로 부여하는 것이 목적임.
{
    struct ashmem_area *asma = file->private_data;
    int ret = 0;

    mutex_lock(&ashmem_mutex);

    /* user needs to SET_SIZE before mapping */
    if (unlikely(!asma->size)) {
        ret = -EINVAL;
        goto out;
    }

    /* requested protection bits must match our allowed protection mask */
    if (unlikely((vma->vm_flags & ~calc_vm_prot_bits(asma->prot_mask)) &
        calc_vm_prot_bits(PROT_MASK))) {
        ret = -EPERM;
        goto out;
    }
    vma->vm_flags &= ~calc_vm_may_flags(~asma->prot_mask);

    if (!asma->file) {    //ashmem 생성 서버 측 코드에서만 수행됨
        char *name = ASHMEM_NAME_DEF;
        struct file *vmfile;
        if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '₩0')
            name = asma->name;    //이름을 변경해 줌.

        vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
            // backing shmem 파일 영역 할당 – 새로운 ashmem 이름을 가진 file 생성
            //tmpfs에 위치하는 unlinked file임
            //이 파일을 상대방 프로세스에 전달하여, mmap_region()에서 할당한 영역을 공유
        if (unlikely(IS_ERR(vmfile))) {
            ret = PTR_ERR(vmfile);
            goto out;
        }
        asma->file = vmfile;
```

```
    }
    get_file(asma->file);

    /*
     * XXX - Reworked to use shmem_zero_setup() instead of
     * shmem_set_file while we're in staging. -jstultz
     */
    if (vma->vm_flags & VM_SHARED) {
        ret = shmem_zero_setup(vma);
        if (ret) {
            fput(asma->file);
            goto out;
        }
    }

    if (vma->vm_file)
        fput(vma->vm_file);
    vma->vm_file = asma->file;

out:
    mutex_unlock(&ashmem_mutex);
    return ret;
}

static const struct file_operations ashmem_fops = {
        .owner = THIS_MODULE,
        .open = ashmem_open,
        .release = ashmem_release,
        .read = ashmem_read,
        .llseek = ashmem_llseek,
        .mmap = ashmem_mmap,
        .unlocked_ioctl = ashmem_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl = compat_ashmem_ioctl,
#endif
};

static int __init ashmem_init(void)
{
```

```
        int ret;

        ashmem_area_cachep = kmem_cache_create("ashmem_area_cache",
                                               sizeof(struct ashmem_area), 0, 0, NULL);
        [...]

        ashmem_range_cachep = kmem_cache_create("ashmem_range_cache",
                                                sizeof(struct ashmem_range), 0, 0, NULL);
        [...]
        register_shrinker(&ashmem_shrinker);
        [...]
        return 0;
}
```

Low Memory Killer 소개

시스템에 더 이상 할당한 페이지가 없을 때, 페이지를 free(shrink라고도 함)해 주는 동작이 필요하다. 리눅스에는 이미 kswapd라는 커널 thread가 이러한 역할을 수행하기 위해 마련되어 있다. kswapd는 평소에는 sleep 상태에 있다가, pages_low에 지정된 값 보다 적은 수의 페이지가 남게 될 경우 깨어나 페이지 수거에 들어간다(mm/page_alloc.c, vmscan.c 파일 참조)
kswapd에서 페이지 수거 작업을 수행하기 위해 호출하는 함수의 흐름을 간략히 정리해 보면 다음과 같다.

<kswapd 함수 호출 흐름>

1) balance_pgdat()
1-1) shrink_zone()
1-1-1) refill_inactive_zone()
1-1-2) shrink_cache()
1-1-2-1) shrink_list()
2) shrink_slab()

위의 코드 중, 마지막에 있는 shrink_slab() 함수는 shrinker_list에 포함된 항목에 대해 loop를 돌면서, shrink 작업을 진행하는데, shrinker_list에 자신을 등록하기 위해서는 아래 함수를 사용해야 한다.

```
static int test_shrink_func(struct shrinker *s, struct shrink_control *sc)
{
        [...]
}
```

```
static struct shrinker test_shrinker = {
    .shrink = test_shrink_func,    //shrink_slab()이 실행될 때, 이 함수가 호출됨.
    .seeks = DEFAULT_SEEKS * 16
};
register_shrinker(&test_shrinker);  //shrinker로 등록
unregister_shrinker(&test_shrinker);  //등록한 shrinker 해제
```

안드로이드 Low Memory Killer는 현재 남아 있는 메모리 량을 측정하여 기준치 이하일 경우, OOM(Out Of Memory) 스코어가 큰 프로세스(task) 부터 차례로 죽여 나가는 기법으로, 메모리 부족 현상을 해결하기 위한 방법으로 볼 수 있다(기존 리눅스의 OOM 기법과 차별화함). 아래 그림은 Low Memory Killer를 그림으로 표현해 본 것으로, 이의 동작 과정을 간략히 정리하면 다음과 같다.

1) 자신을 shrinker(lowmem_shrinker)로 등록한다.

2) 현재 남아 있는 메모리량 측정한다.

```
int other_free = global_page_size(NR_FREE_PAGES);
```

- 전체 Free page의 개수를 구하는 함수
- vm_stat[ ] 배열에 저장된 값을 return하는 구조임.

```
int other_file = global_page_state(NR_FILE_PAGES) – global_page_state(NR_SHMEM);
```

- 전체 Free File Page의 개수를 구하는 함수

3) 남아 있는 메모리 량이 기준 값보다 작을 경우, 해당 adj 값을 리턴한다.

```
if (other_free <   lowmem_minfree[i])
    min_score_adj = lowmem_adj[i];
```

4) 현재 동작 중인 모든 task에 대해, adj 값(oom_score_adj)를 체크하여, 가장 큰 놈을 찾아 낸다.

```
for_each_process(tsk) {
        compare p->signal->oom_score_adj and selected_oom_score_adj
}
```

5) 4)에서 찾은 task를 죽인다(메모리 확보).

```
send_sig(SIGKILL)
```

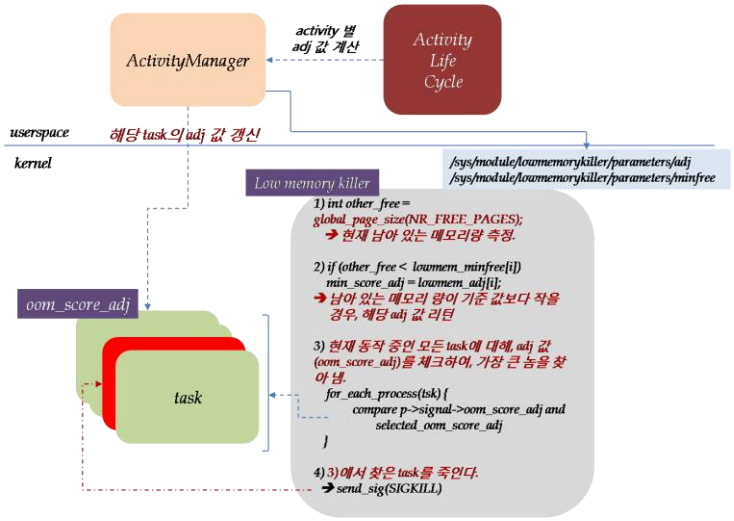


그림 3-22 Low Memory Killer 구조

[예제 코드 - drivers/staging/android/lowmemorykiller.c]

```
static int lowmem_shrink(struct shrinker *s, struct shrink_control *sc)     ②
{
        struct task_struct *tsk;
        struct task_struct *selected = NULL;
        int rem = 0;
        int tasksize;
        int i;
        short min_score_adj = OOM_SCORE_ADJ_MAX + 1;
        int selected_tasksize = 0;
        short selected_oom_score_adj;
        int array_size = ARRAY_SIZE(lowmem_adj);
        int other_free = global_page_state(NR_FREE_PAGES) - totalreserve_pages;
        int other_file = global_page_state(NR_FILE_PAGES) - global_page_state(NR_SHMEM);

        if (lowmem_adj_size < array_size)
                array_size = lowmem_adj_size;
        if (lowmem_minfree_size < array_size)
                array_size = lowmem_minfree_size;
        for (i = 0; i < array_size; i++) {     //조건 검사     ③
                if (other_free < lowmem_minfree[i] && other_file < lowmem_minfree[i]) {
                        min_score_adj = lowmem_adj[i];
                        break;
                }
```

```
    }
    if (sc->nr_to_scan > 0)
        lowmem_print(3, "lowmem_shrink %lu, %x, ofree %d %d, ma %hd₩n",
                    sc->nr_to_scan, sc->gfp_mask, other_free, other_file, min_score_adj);
    rem = global_page_state(NR_ACTIVE_ANON) + global_page_state(NR_ACTIVE_FILE) +
        global_page_state(NR_INACTIVE_ANON) + global_page_state(NR_INACTIVE_FILE);
    if (sc->nr_to_scan <= 0 || min_score_adj == OOM_SCORE_ADJ_MAX + 1) {
        lowmem_print(5, "lowmem_shrink %lu, %x, return %d₩n",
                    sc->nr_to_scan, sc->gfp_mask, rem);
        return rem;
    }
    selected_oom_score_adj = min_score_adj;     //OOM score adj 값 구함     ④

    rcu_read_lock();     //RCU lock 진입
    for_each_process(tsk) {
      //모든 task에 대해 반복하면서, oom_score_adj가 가장 큰 task를 찾음.     ⑤
        struct task_struct *p;
        short oom_score_adj;
        if (tsk->flags & PF_KTHREAD)
            continue;

        p = find_lock_task_mm(tsk);
        if (!p)
            continue;

        if (test_tsk_thread_flag(p, TIF_MEMDIE) &&
                        time_before_eq(jiffies, lowmem_deathpending_timeout)) {
            task_unlock(p);
            rcu_read_unlock();
            return 0;
        }
        oom_score_adj = p->signal->oom_score_adj;
        if (oom_score_adj < min_score_adj) {
            task_unlock(p);
            continue;
        }
        tasksize = get_mm_rss(p->mm);
        task_unlock(p);
```

```
                if (tasksize <= 0)
                        continue;
                if (selected) {
                        if (oom_score_adj < selected_oom_score_adj)
                                continue;
                        if (oom_score_adj == selected_oom_score_adj &&
                          tasksize <= selected_tasksize)
                                continue;
                }
                selected = p;
                selected_tasksize = tasksize;
                selected_oom_score_adj = oom_score_adj;
                lowmem_print(2, "select %d (%s), adj %hd, size %d, to kill₩n",
                                p->pid, p->comm, oom_score_adj, tasksize);
        }
        if (selected) {     //찾은 task를 죽임(SIGKILL) ⑥
                lowmem_print(1, "send sigkill to %d (%s), adj %hd, size %d₩n",
                                selected->pid, selected->comm,
                                selected_oom_score_adj, selected_tasksize);
                lowmem_deathpending_timeout = jiffies + HZ;
                send_sig(SIGKILL, selected, 0);
                set_tsk_thread_flag(selected, TIF_MEMDIE);
                rem -= selected_tasksize;
        }
        lowmem_print(4, "lowmem_shrink %lu, %x, return %d₩n",
                        sc->nr_to_scan, sc->gfp_mask, rem);
        rcu_read_unlock();     //RCU lock 해제
        return rem;
}

static struct shrinker lowmem_shrinker = {     //shrinker 선언
        .shrink = lowmem_shrink,
        .seeks = DEFAULT_SEEKS * 16
};

static int __init lowmem_init(void)
{
        register_shrinker(&lowmem_shrinker);     //shrinker로 등록     ①
```

```
        return 0;
}
```

4. 통합 Device 모델과 Sysfs

이번 장에서 설명하는 통합 디바이스 모델은 앞으로 설명할 3장 ~ 6장의 내용과도 밀접히 연관되어 있는 매우 중요한 내용이라 할 수 있다.

통합 Device 모델

통합 디바이스 모델(장치 및 장치 드라이버에 관한 모델)은 커널 2.6에서부터 사용되기 시작하였다. 통합 디바이스 모델을 사용하면서, 모든 장치는 동일한 데이터 구조와 함수를 사용하는 하나의 통일된 프레임워크로 처리되게 되었다. 통합 디바이스 모델을 위한 데이터 구조에는 주로 아래와 같은 정보가 담겨 있다.

1) 장치를 위해 어떠한 드라이버가 사용되고 있는지 ..
2) 장치가 어떤 버스 상에 놓여 있는지 ...
3) 장치의 파워 상태는 어떤지 ...
4) 장치가 어떻게 suspend 혹은 resume이 되는지 ..

코드 2-x include/linux/device.h

```
struct bus_type {
        const char          *name;
        const char          *dev_name;
        struct device           *dev_root;
        struct bus_attribute        *bus_attrs;
        struct device_attribute *dev_attrs;
        struct driver_attribute *drv_attrs;

        int (*match)(struct device *dev, struct device_driver *drv);
        int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);

        int (*suspend)(struct device *dev, pm_message_t state);
        int (*resume)(struct device *dev);
```

```
	const struct dev_pm_ops *pm;

	struct iommu_ops *iommu_ops;

	struct subsys_private *p;
	struct lock_class_key lock_key;
};

struct device_driver {
	const char		*name;
	struct bus_type		*bus;

	struct module		*owner;
	const char		*mod_name;	/* used for built-in modules */

	bool suppress_bind_attrs;	/* disables bind/unbind via sysfs */

	const struct of_device_id	*of_match_table;
	const struct acpi_device_id *acpi_match_table;

	int (*probe) (struct device *dev);
	int (*remove) (struct device *dev);
	void (*shutdown) (struct device *dev);
	int (*suspend) (struct device *dev, pm_message_t state);
	int (*resume) (struct device *dev);
	const struct attribute_group **groups;

	const struct dev_pm_ops *pm;

	struct driver_private *p;
};

struct device {
	struct device		*parent;

	struct device_private	*p;

	struct kobject kobj;
	const char		*init_name; /* initial name of the device */
```

```
    const struct device_type *type;

    struct mutex            mutex;  /* mutex to synchronize calls to its driver. */

    struct bus_type *bus;           /* type of bus device is on */
    struct device_driver *driver;   /* which driver has allocated this device */
    void            *platform_data; /* Platform specific data, device core doesn't touch it */
    struct dev_pm_info  power;
    struct dev_pm_domain    *pm_domain;

#ifdef CONFIG_PINCTRL
    struct dev_pin_info *pins;
#endif

#ifdef CONFIG_NUMA
    int     numa_node;  /* NUMA node this device is close to */
#endif
    u64     *dma_mask;  /* dma mask (if dma'able device) */
    u64     coherent_dma_mask;/* Like dma_mask, but for
                                 alloc_coherent mappings as
                                 not all hardware supports
                                 64 bit addresses for consistent
                                 allocations such descriptors. */

    struct device_dma_parameters *dma_parms;

    struct list_head    dma_pools;  /* dma pools (if dma'ble) */

    struct dma_coherent_mem *dma_mem; /* internal for coherent mem override */
#ifdef CONFIG_CMA
    struct cma *cma_area;           /* contiguous memory area for dma allocations */
#endif
    /* arch specific additions */
    struct dev_archdata archdata;

    struct device_node  *of_node; /* associated device tree node */
    struct acpi_dev_node    acpi_node; /* associated ACPI device node */

    dev_t               devt;   /* dev_t, creates the sysfs "dev" */
```

```
    u32             id; /* device instance */

    spinlock_t      devres_lock;
    struct list_head    devres_head;

    struct klist_node   knode_class;
    struct class        *class;
    const struct attribute_group **groups;   /* optional groups */

    void    (*release)(struct device *dev);
    struct iommu_group  *iommu_group;
};
```

시스템 코어에(로 부터) 장치(struct device)를 등록하거나, 해제하기 위해서는 아래의 함수를 사용한다.

int ***device_register****(struct device *dev)*
void ***device_unregister****(struct device *dev)*

- drivers/base/core.c에 정의되어 있음.

한편 드라이버(struct device_driver)를 등록하거나, 해제하기 위해서는 아래의 함수를 사용한다.

int ***driver_register****(struct device_driver *drv)*
void ***driver_unregister****(struct device_driver *drv)*

- drivers/base/driver.c에 정의되어 있음.

그런데, 일반적으로 실제 디바이스 드라이버(예: USB, I^2C, SPI 등) 의 경우에는 위에서 기술한 데이터 구조를 직접 사용한다기 보다는, 특정 유형의 버스나 장치를 위해 자체적으로 정의한 코드 안에서 사용하는 형태를 띈다. 아래 pseudo 코드를 참고하기 바란다.

a) real_dev structure와 real_driver structure 선언

```
struct real_dev {
    [...]
    struct real_driver *driver;
    [...]
    struct device dev;
    [...]
}
```

```
struct real_driver {
    [...]
    struct device_driver driver;
    [...]
}
```

b) Driver는 아래 함수를 써서 등록 혹은 해제한다.

```
#include <linux/real.h>
int real_register_driver (struct real_driver *);
void real_unregister_driver(struct real_driver *);
```

c) 한편, Device는 probe() 함수 내에서 찾은 후, 등록된다.

d) 아래 코드는 device structure로부터 real_dev structure 찾아내는 것을 보여준다.

```
#define to_real_dev(n)    container_of(n, struct real_dev, dev)
struct device *dev;
[...]
struct real_dev *pdev = to_real_dev (dev);
```

e) 마지막으로 아래 코드는 device_driver structure로부터 real_driver structure를 찾아내는 것을 보여준다.

```
struct device_driver *drv;
[...]
struct real_driver *pdrv = to_real_drv (drv);
```

버스 드라이버

버스 드라이버는 통합 디바이스 모델의 핵심 요소이다. 각각의 버스 형식(USB, PCI, SPI, MMC, I2C 등)에 대해 서로 다른 버스 드라이버가 존재하게 되는데, 각각의 버스 드라이버가 하는 일을 정리하면 다음과 같다.

1) bus type structure를 사용하여 버스 유형을 등록해준다.
2) 어댑터 혹은 인터페이스(USB controllers, I2C controllers, SPI controllers) 드라이버를 등록해준다. 어댑터 혹은 인터페이스(혹은 컨트롤러)는 버스에 연결된 장치를 감지하거나 제어하는 하드웨어 장치를 말한다.
3) 실제 장치(USB devices, I2C devices, SPI devices) 드라이버를 등록한다.
4) 감지한 장치와 실제 장치 드라이버를 연결시켜준다.

그림 3-23은 어댑터(혹은 컨트롤러), 버스, 장치(장치 드라이버) 간의 관계를 USB를 예로 하여 하나로 그려본 것이다.

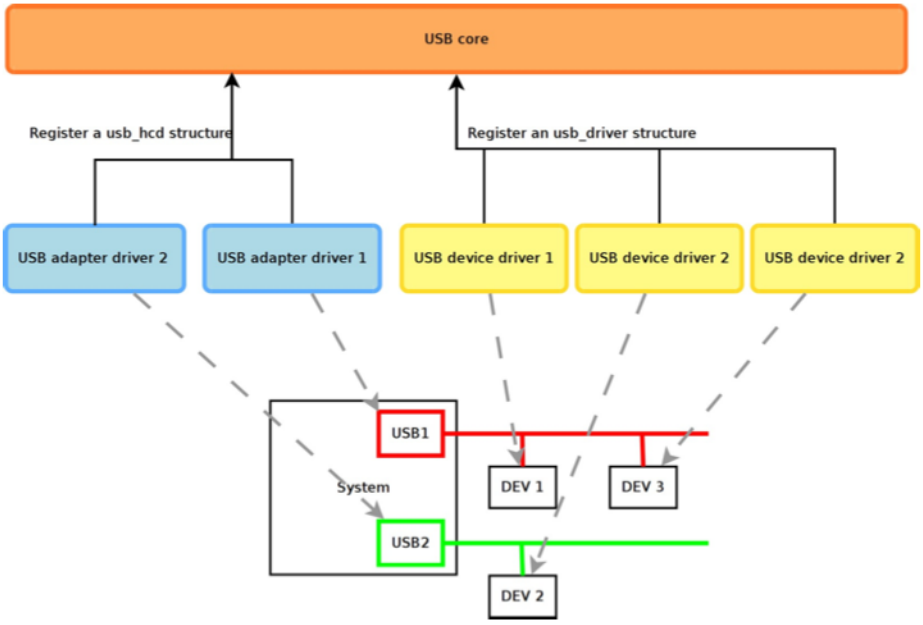


그림 3-23 어댑터, 버스, 디바이스 드라이버의 관계 – USB 예 [출처 – 참고 문헌 8]

장치 드라이버를 등록 후 하게 되는 최초의 일이 장치 드라이버의 probe() 함수를 호출하는 일인데, 이를 수행하기까지의 과정을 3단계로 나누어 살펴 보면 다음과 같다.

1) 부팅 시, USB 장치 드라이버는 자신을 USB 버스 드라이버로 등록한다.

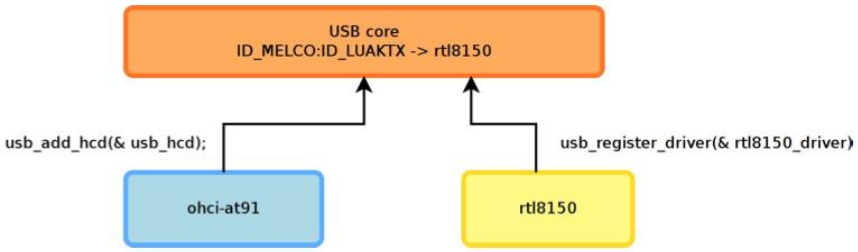


그림 3-24 probe() 함수 호출 과정(1) [출처 – 참고 문헌 8]

2) USB 버스 어댑터 드라이버가 USB 장치를 감지하면, USB 코어(혹은 USB bus infrastructure)에 이를 알린다.

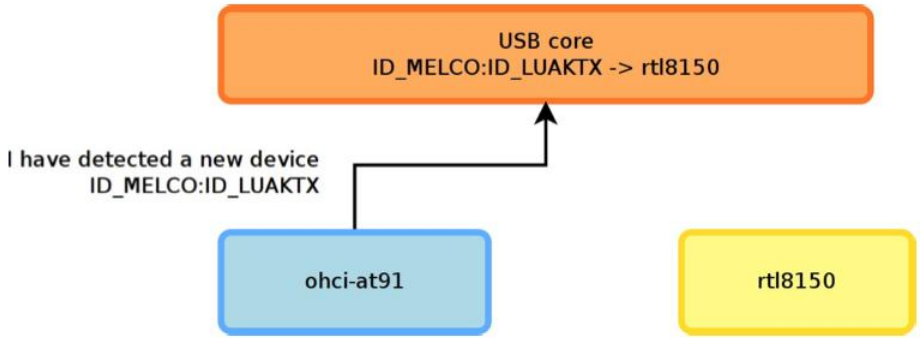


그림 3-25 probe() 함수 호출 과정(2) [출처 – 참고 문헌 8]

3) USB core(USB bus infrastructure)는 어느 드라이버가 방금 감지한 장치를 다룰 수 있는지를 알고 있으므로, 해당 드라이버의 probe() 함수를 호출한다.

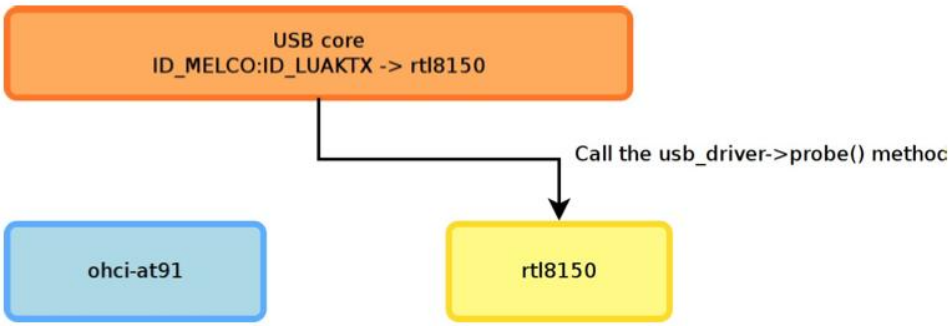


그림 3-26 probe() 함수 호출 과정(3) [출처 – 참고 문헌 8]

Probe() 함수는 장치를 묘사하는 structure(예: pci dev, usb interface 등)를 파라미터로 전달 받게 된다. Probe() 함수에서 하는 일은 다음과 같다.

1) 장치를 초기화 하고, I/O 메모리를 매핑시켜 주며, 인터럽트 핸들러를 등록한다. 한편 버스 코어(infrastructure)는 장치 주소, IRQ 번호 및 장치 고유의 정보를 얻을 수 있는 방법을 제공하므로, 이를 사용한다.
2) 장치를 적당한 커널 프레임워크(혹은 서브 시스템)에 등록한다.

<최근 커널 디바이스 드라이버 Probe 함수 해부>

1) *Device model에 기초한 코드*
2) *Platform device or Device Tree 기반의 device 정보* *<3장에서 소개>*
3) *메모리 버퍼 할당*
4) *Interrupt handler 등록*
5) *Work queue 선언*
6) *Kernel thread 선언*
7) *Notifier chain 선언*
8) *Synchronization 함수 사용(spinlock, mutex, completion 등)*

9) *Sysfs 초기화 코드*

그림 3-27은 커널에서 제공하는 다양한 프레임워크를 그림으로 표현한 것이다.

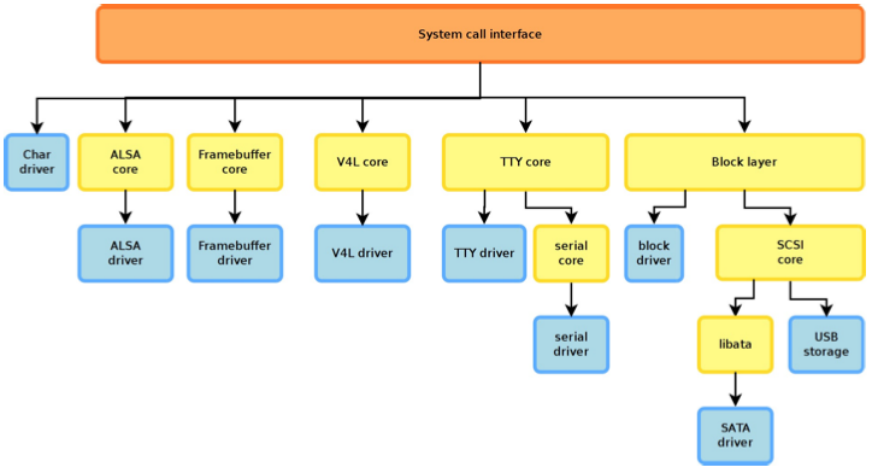


그림 3-27 커널 프레임워크 개요 [출처 – 참고 문헌 8]

지금까지 설명한 내용을 정리해 보면 다음과 같다.

1) 커널 2.6부터는 device 및 device driver를 하나로 관리하는 통합 디바이스 모델이 새롭게 구현되었다.
2) 이는 주로 bus 형태의 어댑터 및 장치를 위해 만들어진 것으로 보아야 한다.
3) Bus의 종류는 여러 가지가 있으며, 각각의 장치를 효과적으로 관리하기 위해 자신만의 고유한 프레임워크를 구현해 놓고 있다.
4) 따라서, 개별 장치 드라이버를 이러한 프레임워크를 기반으로 보다 쉽게 장치 드라이버를 구현할 수가 있다.

마지막으로, 임베디드 시스템의 경우에는 장치들이 버스를 통해 연결되지 않고 바로 CPU에 연결되는 형태를 취하고 있다. 리눅스에서는 이러한 장치에 대해서도, 지금까지 설명한 통합 디바이스 모델을 따르기를 원했다. 따라서 새로운 개념이 필요했는데, 이것이 바로 platform device/driver 프레임워크이다. 이에 관해서는 3장에서 자세히 살펴 보기로 하겠다.

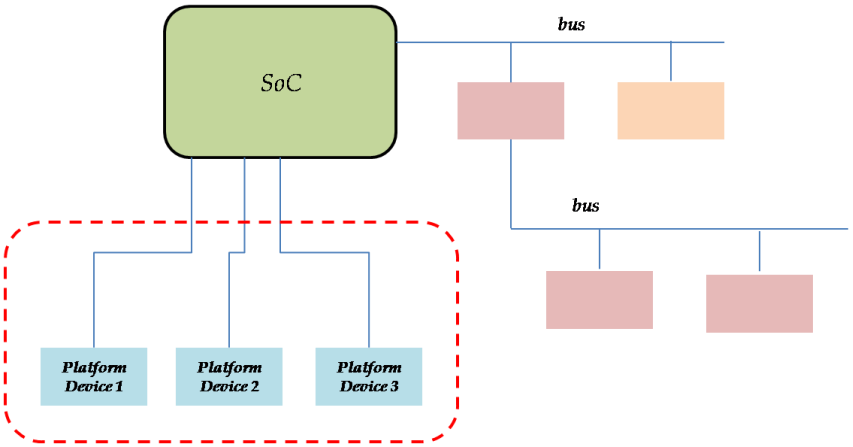


그림 3-28 Platform Device 개념(1)

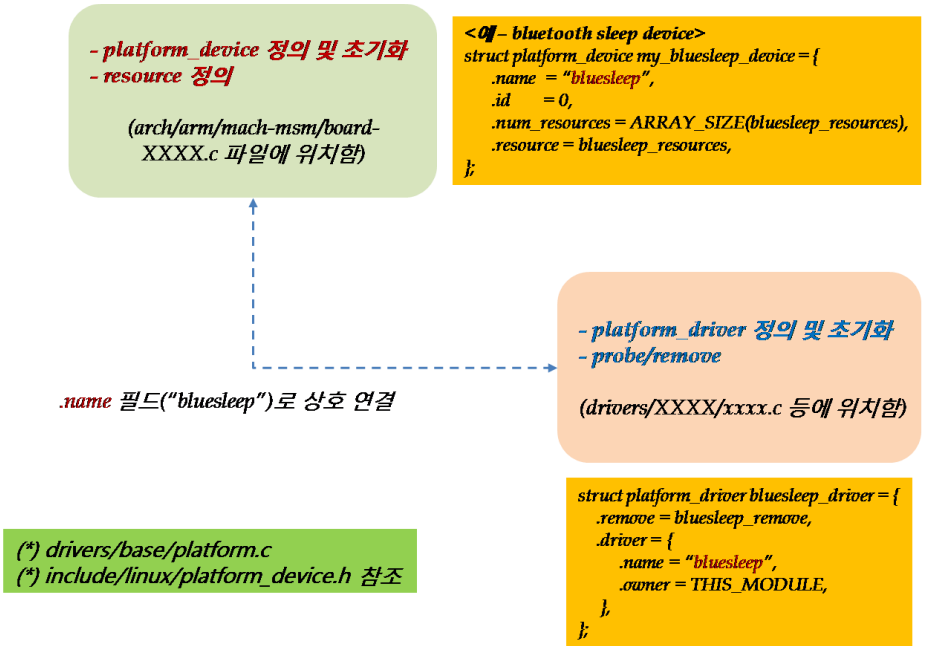


그림 3-29 Platform Device 개념(2)

kobject와 sysfs

kobject(kernel object)는 device model을 위해 등장한 것으로, kset은 kobject의 묶음이고, subsystem은 kset의 묶음이다. sysfs는 kobject의 계층 tree를 표현(view)해 주는 memory 기반의 file system으로 kernel 2.6에서부터 소개되었다. kernel device와 user process가 소통(통신)하는 수단 중 하나로 볼 수 있으며, 이와 유사한 것으로 proc file system 등이 있다.

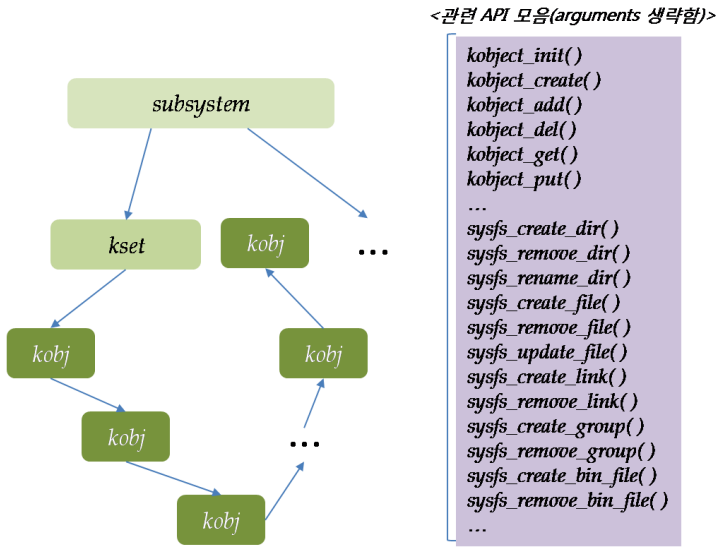


그림 3-30 kobject와 sysfs 개념 이해(1)

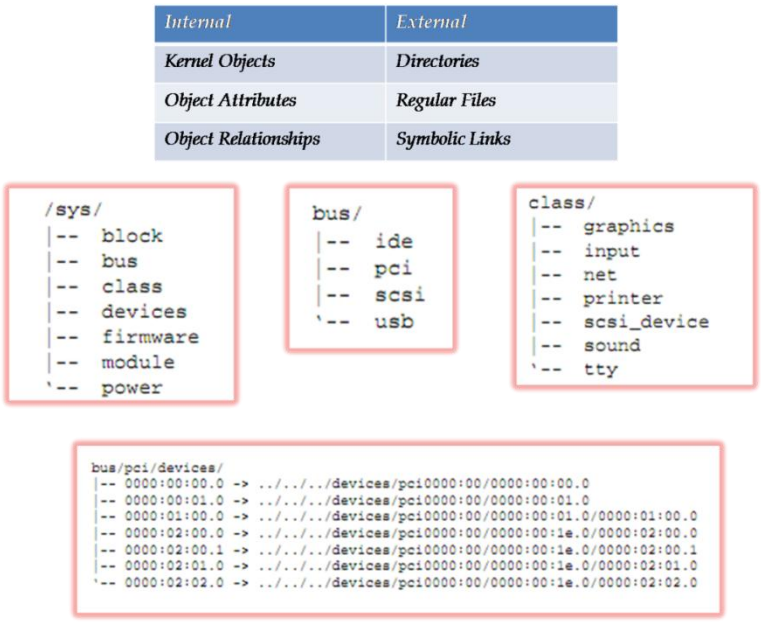
코드 2-x include/linux/kobject.h

```
struct kobject {
        const char          *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset         *kset;
        struct kobj_type        *ktype;
        struct sysfs_dirent *sd;
        struct kref         kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
[...]
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
};
```

```
struct kobj_uevent_env {
        char *envp[UEVENT_NUM_ENVP];
        int envp_idx;
        char buf[UEVENT_BUFFER_SIZE];
        int buflen;
};

struct kset_uevent_ops {
        int (* const filter)(struct kset *kset, struct kobject *kobj);
        const char *(* const name)(struct kset *kset, struct kobject *kobj);
        int (* const uevent)(struct kset *kset, struct kobject *kobj,
                             struct kobj_uevent_env *env);
};

struct kobj_attribute {
        struct attribute attr;
        ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf);
        ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
                         const char *buf, size_t count);
};
[...]
struct kset {
        struct list_head list;
        spinlock_t list_lock;
        struct kobject kobj;
        const struct kset_uevent_ops *uevent_ops;
};
```



Internal	External
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

```
/sys/
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- module
`-- power
```

```
bus/
|-- ide
|-- pci
|-- scsi
`-- usb
```

```
class/
|-- graphics
|-- input
|-- net
|-- printer
|-- scsi_device
|-- sound
`-- tty
```

```
bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
|-- 0000:01:00.0 -> ../../../devices/pci0000:00/0000:00:01.0/0000:01:00.0
|-- 0000:02:00.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.0
|-- 0000:02:00.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.1
|-- 0000:02:01.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
`-- 0000:02:02.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:02.0
```

그림 3-31 kobject와 sysfs 개념 이해(2)

앞서 언급한 kobject_ 및 sysfs_ API를 이용하여 직접 작업하는 것도 가능하나, 보다 편리한 방법으로 다음과 같은 API의 사용이 가능하다(앞서 제시한 API를 사용할 경우, 매우 세세한 제어가 가능할 것임). 드라이버 초기화 시, device_create_file( )을 통해 sysfs 파일 생성이 가능하며, 드라이버 제거 시, device_remove_file( )을 통해 만들어 둔, sysfs 파일이 제거된다. device_create_file( )로 만들어둔, file을 읽고, 쓸 경우에는 각각 show 및 store에 정의한 함수가 불리어질 것이다. 참고로 platform device의 경우에는, device_create_file의 첫 번째 argument 값으로 .dev 필드의 정보가 전달되어야 한다.

```
struct device_attribute {
  struct attribute        attr;
  ssize_t (*show)(struct device *dev, char *buf);
  ssize_t (*store)(struct device *dev, const char *buf, size_t count);
};

int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

그림 3-32 kobject와 sysfs 개념 이해(3) – include/linux/device.h

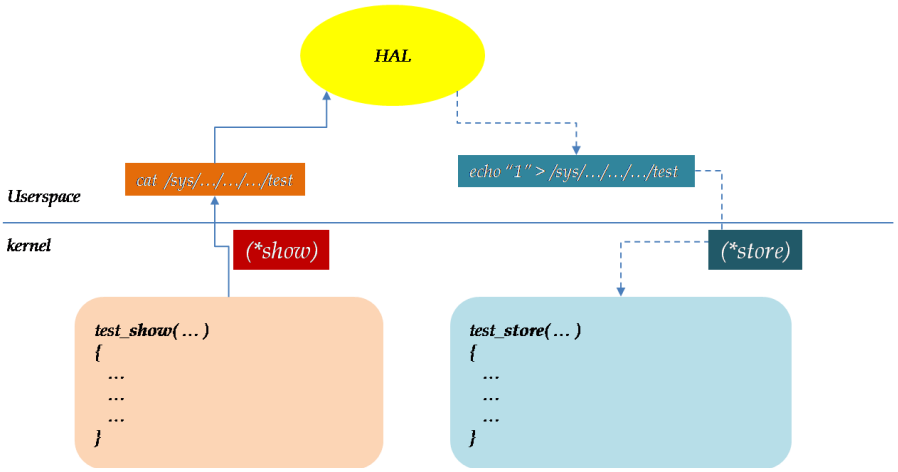


그림 3-33 sysfs operations – show, store

[예제 코드 - drivers/leds/trigger/ledtrig-backlight.c]

```
static ssize_t bl_trig_invert_show(struct device *dev, struct device_attribute *attr, char *buf)
  // 사용자 영역의 프로그램에서 sysfs의 내용을 보고자 할 경우 사용(get operation) ④ or ⑤
{
    struct led_classdev *led = dev_get_drvdata(dev);
    struct bl_trig_notifier *n = led->trigger_data;
    return sprintf(buf, "%u₩n", n->invert);
}

static ssize_t bl_trig_invert_store(struct device *dev, struct device_attribute *attr, const char *buf,
size_t num)
    // 사용자 영역의 프로그램에서 sysfs에 값을 쓰고자 할 경우에 사용(set operation) ④ or ⑤
{
    struct led_classdev *led = dev_get_drvdata(dev);
    struct bl_trig_notifier *n = led->trigger_data;
    unsigned long invert;
    int ret;

    [...]

    /* After inverting, we need to update the LED. */
    if ((n->old_status == BLANK) ^ n->invert)
        __led_set_brightness(led, LED_OFF);
    else
        __led_set_brightness(led, n->brightness);
```

```
    return num;
}

static DEVICE_ATTR(inverted, 0644, bl_trig_invert_show, bl_trig_invert_store);     ①
     // 아래 매크로임
     // #define DEVICE_ATTR(_name, _mode, _show, _store) ₩
     // struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)

static void bl_trig_activate(struct led_classdev *led)     ②
{
    [...]
    ret = device_create_file(led->dev, &dev_attr_inverted);     ③
        //led->dev 장치에 대한 sysfs attribute 파일을 생성해 줌. /sys/devices 아래에 파일 생성됨.
    if (ret)
        goto err_invert;
    [...]
}

static void bl_trig_deactivate(struct led_classdev *led)     ⑥
{
    [...]
    device_remove_file(led->dev, &dev_attr_inverted);     ⑦
        //led->dev 장치에 대한 sysfs attribute 파일을 제거해 줌. /sys/devices 아래에 파일 제거
됨.

    [...]
}
```

kobject로 할 수 있는 또 다른 일로는, 커널로부터 사용자 계층으로 event를 전달하는 것이다. 아래 그림은 디바이스 드라이버와 ueventd 프로세스가 kobject_uevent( ) 함수에 의해 상호 통신하는 그림으로, uevent를 받은 ueventd가 새로운 device를 sysfs 파일 시스템에 생성하는 것으로 보여주고 있다. 아래 그림에서도 확인 가능하듯이 kobject_uevent() 함수는 netlink socket을 사용하여 구현되어 있으므로, socket을 통해 손쉽게 사용자 계층으로 uevent 전달이 가능해진다.

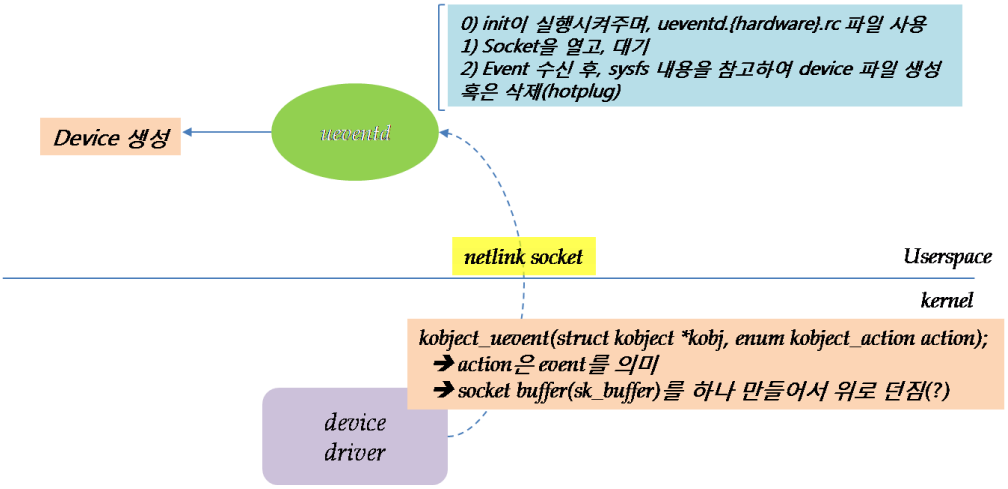


그림 3-34 kobject와 sysfs 개념 이해(4) – ueventd와 sysfs

5. 기타 참고 사항

끝으로, 본서에서는 아래와 같은 내용은 특별히 설명의 대상에서 제외하였다. 혹시 관련하여 궁금한 사항이 있는 독자들은 참고 문헌을 참고하기 바란다.

1) Character, Block, Network 등 기본 디바이스 드라이버

- *file operations – open/read/write/ioctl ..*
- *User space에서의 접근 방법*

2) Proc file system

3) System call

4) Kernel module 생성 및 초기화

5) Kernel configuration 제어 관련

6) kernel data structure(linked list, queue, rbtree ...)

➔ 여기까지는 기초적인 부분이라 설명에서 배제하였음.

7) Hardware I/O

8) Memory management 기법 중 상세 부분

9) Page & Buffer cache 기법

10) 상세한 task(process) 관리 기법

11) 파일 시스템 관리 기법

➔ 이 부분은 본 서의 취지에서 벗어나는 부분으로 설명에서 제외하였음.

본 서의 내용 중 일부는 Free Electrons의 문서[참고문헌 8-10]를 참고하였음을 밝힌다. 특히 그림 중 일부는 그대로 복사하여 사용하였다.

References

[1] Linux Kernel Development(3rd edition), Robert Love, Addison Wesley

[2] Writing Linux Device Drivers, Jerry Cooperstein

[3] Essential Linux Device Drivers, Sreekrishnan Venkateswaran, Prentice Hall

[4] Linux kernel 2.6 구조와 원리, 이영희 역, 한빛미디어

[5] 코드로 알아보는 ARM 리눅스 커널, 노서영 외 5인, Jpub

[6] 리눅스 커널 내부 구조, 백승재, 최종무, 교학사

[7] Professional Linux Kernel Architecture, Wolfgang Mauerer, Wrox

[8] Understanding Linux Kernel(3rd endition), Daniel P. Bovet, Marco Cesati, O'Reilly

[9] Android Kernel Hacks1/2, Chunghan Yi, www.kandroid.org

[10] Linux Kernel and Driver Development Training, Gregory Clement, Michael Opdenacker, Maxime Ripard, S_ebastien Jan, Thomas Petazzoni, Free Electrons

[11] Embedded Linux system development, Gregory Clement, Michael Opdenacker, Maxime Ripard, Thomas Petazzoni, Free Electrons

[12] Linux Kernel architecture for device drivers, Thomas Petazzoni Free Electronics

[13] The sysfs Filesystem, Patrick Mochel, mochel@digitalimplant.org

[14] InterruptThreads-Slides_Anderson.pdf, Mike Anderson, mike@theptrgroup.com