(1) in kernel/sched/core.c

```
/*
 * this is the entry point to schedule() from in-kernel preemption
 * off of preempt_enable. Kernel preemptions off return from interrupt
 * occur there and call schedule directly.
 */
asmlinkage __visible void __sched notrace preempt_schedule(void)
{
	/*
	 * If there is a non-zero preempt_count or interrupts are disabled,
	 * we do not want to preempt the current task. Just return..
	 */
	if (likely(!preemptible()))                    (1)
		return;

	do {
		__preempt_count_add(PREEMPT_ACTIVE);
		__schedule();
		__preempt_count_sub(PREEMPT_ACTIVE);

		/*
		 * Check again in case we missed a preemption opportunity
		 * between schedule and now.
```

```
        */
        barrier();
    } while (need_resched());
}
```

(1)

先判断当前是否可抢占，若否，则直接return;若可以，则发生context switch.

Question:是否可抢占的标准是什么呢？

__schedule()是context switch的core function.

```
#ifdef CONFIG_PREEMPT_COUNT
# define preemptible()   (preempt_count() == 0 && !irqs_disabled())
#else
# define preemptible()   0
#endif
```

in Gs2 / Gr2 LSP,

```
CONFIG_NR_CPUS=4
CONFIG_HOTPLUG_CPU=y
CONFIG_PREEMPT=y
CONFIG_PREEMPT_COUNT=y
```

有4颗A53,non-preemptible是说不过去的。

irqs_disabled()就是去获得当前系统的irq是否被disable.如果处于disabled，当然是不允许发生context switch。这一般只有在hardware interrupt的关键阶段才发生

```
static __always_inline int preempt_count(void)

{

        return current_thread_info()->preempt_count;

}


/*

 * how to get the thread information struct from C

 */

static inline struct thread_info *current_thread_info(void) __attribute_const__;


static inline struct thread_info *current_thread_info(void)

{

        register unsigned long sp asm ("sp");

        return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));

}
```

register "sp"指向current kernel stack.

sp & ~(THREAD_SIZE - 1)

#define THREAD_SIZE_ORDER 1

#define THREAD_SIZE (PAGE_SIZE << THREAD_SIZE_ORDER)

#define THREAD_START_SP (THREAD_SIZE - 8)

THEAD_SIZE = kernel stack size = PAGE_SIZE << 1 = 8K (保留了栈顶的8个bytes)

kernel stack从这8K空间的尾部往下增长(stack 由高地址往低地址增长)

kernel stack的头上是thread_info structure.

(struct thread_info *)(sp & ¯(THREAD_SIZE - 1));

取得current thread info.

```
struct thread_info {
	unsigned long		flags;		/* low level flags */
	int			preempt_count;	/* 0 => preemptable, <0 => bug */
	mm_segment_t		addr_limit;	/* address limit */
	struct task_struct	*task;		/* main task structure */
	struct exec_domain	*exec_domain;	/* execution domain */
	__u32			cpu;		/* cpu */
__u32			cpu_domain;	/* cpu domain */
	struct cpu_context_savecpu_context;	/* cpu context */
	__u32			syscall;	/* syscall number */
	__u8			used_cp[16];	/* thread used copro */
	unsigned long		tp_value[2];	/* TLS registers */
#ifdef CONFIG_CRUNCH
```

```
        struct crunch_state crunchstate;
#endif
        union fp_state          fpstate __attribute__((aligned(8)));
        union vfp_state                 vfpstate;
#ifdef CONFIG_ARM_THUMBEE
        unsigned long           thumbee_state;      /* ThumbEE Handler Base register */
#endif
        struct restart_block restart_block;
};
```

所以preempt_count()就是返回current thread_info structure中的preempt_count field.如果该field非零，表示目前不允许发生抢占，即不允许context switch。

在ARM interrupt handler完毕后，都会check是否可以发生抢占。

比如当kernel从interrupt返回时

in arch/arm/kernel/entry-armv.S

```
        .align  5
__irq_svc:
        svc_entry
        irq_handler

#ifdef CONFIG_PREEMPT
        get_thread_info tsk            （1）
        ldr     r8, [tsk, #TI_PREEMPT]        @ get preempt count            （2）
        ldr     r0, [tsk, #TI_FLAGS]          @ get flags
        teq     r8, #0                        @ if preempt count != 0        （3）
        movne   r0, #0                        @ force flags to 0
        tst     r0, #_TIF_NEED_RESCHED
        blne    svc_preempt
#endif

        svc_exit r5, irq = 1                  @ return from exception
 UNWIND(.fnend          )
ENDPROC(__irq_svc)
```

（1）

tsk中包含current thread_info structure pointer

（2）

r8 = thread_into->preempt_count

（3）

compair r8 with 0

在interrupt返回时通过判断current thread_info的preempt_count来决定是否允许发生抢占。

由于目前的Linux kernel已经允许在内核态也可以发生抢占的（最起码是部分可抢占，而且可抢占的粒度越来越细。而内核态任意可抢占，应该是无法达到的，最起码这超出了我的理解能力），所以内核态programming确实难度比以前加大了（较早以前在内核态是不可抢占的），即kernel programmer唯一要关心的抢占可能就是interrupt handler可能抢占你的代码，但现在要关心的就多了。

struct thread_info与struct task_struct是什么关系呢？