thread在状态变化时会发出notification,其他kernel code可以向thread注册并等待这些notification.

```
#include <asm/thread_notify.h>
```

notification种类

```
/*
 * These are the reason codes for the thread notifier.
 */
#define THREAD_NOTIFY_FLUSH 0
#define THREAD_NOTIFY_EXIT  1
#define THREAD_NOTIFY_SWITCH    2
#define THREAD_NOTIFY_COPY  3
```

in arch/arm/kernel/process.c

"flush"不知道是什么意思?

```
void flush_thread(void)
{
    struct thread_info *thread = current_thread_info();
    struct task_struct *tsk = current;

    flush_ptrace_hw_breakpoint(tsk);

    memset(thread->used_cp, 0, sizeof(thread->used_cp));
    memset(&tsk->thread.debug, 0, sizeof(struct debug_info));
    memset(&thread->fpstate, 0, sizeof(union fp_state));

    flush_tls();

    thread_notify(THREAD_NOTIFY_FLUSH, thread);
}
```

线程结束时

```
/*
 * Free current thread data structures etc..
 */
void exit_thread(void)
{
    thread_notify(THREAD_NOTIFY_EXIT, current_thread_info());
}
```

复制thread(也就是thread创建时)

```
int
copy_thread(unsigned long clone_flags, unsigned long stack_start,
        unsigned long stk_sz, struct task_struct *p)
{

    ......

    thread_notify(THREAD_NOTIFY_COPY, thread);
}
```

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

发生thread switch时

```
/*
 * Register switch for ARMv3 and ARMv4 processors
 * r0 = previous task_struct, r1 = previous thread_info, r2 = next thread_info
 * previous and next are guaranteed not to be the same.
 */
ENTRY(__switch_to)
......

    mov r5, r0
    add r4, r2, #TI_CPU_SAVE
    ldr r0, =thread_notify_head
    mov r1, #THREAD_NOTIFY_SWITCH
    bl  atomic_notifier_call_chain

......
```

显然这是在发生context switch时发出的。

在支持floating point运算时，使用到了这个机制。

in arch/arm/vfp/vfpmodule.c

> thread_register_notifier(&vfp_notifier_block);

kernel不支持核心态的浮点运算，但用户态application是必须支持的。

```
thread_register_notifier(&vfp_notifier_block);
```

```c
/*
 * When this function is called with the following 'cmd's, the following
 * is true while this function is being run:
 *  THREAD_NOFTIFY_SWTICH:
 *   - the previously running thread will not be scheduled onto another CPU.
 *   - the next thread to be run (v) will not be running on another CPU.
 *   - thread->cpu is the local CPU number
 *   - not preemptible as we're called in the middle of a thread switch
 *  THREAD_NOTIFY_FLUSH:
 *   - the thread (v) will be running on the local CPU, so
 *  v === current_thread_info()
 *   - thread->cpu is the local CPU number at the time it is accessed,
 *  but may change at any time.
 *   - we could be preempted if tree preempt rcu is enabled, so
 *  it is unsafe to use thread->cpu.
 *  THREAD_NOTIFY_EXIT
 *   - the thread (v) will be running on the local CPU, so
 *  v === current_thread_info()
 *   - thread->cpu is the local CPU number at the time it is accessed,
 *  but may change at any time.
 *   - we could be preempted if tree preempt rcu is enabled, so
 *  it is unsafe to use thread->cpu.
 */
static int vfp_notifier(struct notifier_block *self, unsigned long cmd, void *v)
{
    struct thread_info *thread = v;
    u32 fpexc;
#ifdef CONFIG_SMP
    unsigned int cpu;
#endif

    switch (cmd) {
    case THREAD_NOTIFY_SWITCH:
        fpexc = fmrx(FPEXC);

#ifdef CONFIG_SMP
        cpu = thread->cpu;

        /*
         * On SMP, if VFP is enabled, save the old state in
         * case the thread migrates to a different CPU. The
         * restoring is done lazily.
         */
        if ((fpexc & FPEXC_EN) && vfp_current_hw_state[cpu])
            vfp_save_state(vfp_current_hw_state[cpu], fpexc);
#endif

        /*
         * Always disable VFP so we can lazily save/restore the
         * old state.
         */
        fmxr(FPEXC, fpexc & ~FPEXC_EN);
```

```
            break;

        case THREAD_NOTIFY_FLUSH:
            vfp_thread_flush(thread);
            break;

        case THREAD_NOTIFY_EXIT:
            vfp_thread_exit(thread);
            break;

        case THREAD_NOTIFY_COPY:
            vfp_thread_copy(thread);
            break;
        }

        return NOTIFY_DONE;
}
```