MIPS Linux异常中断代码分析

CurrentVersion: 0.16
Date: 2007-04-12
Author: Dajie Tan <jiankemeng@gmail.com>

版本历史

版本状态	作 者	参与者	起止日期	备注
0.10	Dajie Tan		07-04-12	完成草稿
0.12	Dajie Tan		07-04-19	完善 A
0.14	Dajie Tan		07-08-17	完善 B.1
0.15	Dajie Tan		07-08-19	完善 B.2
0.16	Dajie Tan		07-09-10	完善 B.3

本文拟以龙芯 2E （兼容 MIPS III）为例，对内核的异常、中断系统作一个框架式的描述，将结合 2.6.18 的代码来说明。

A. 概述

龙芯 2E 在高优先级异常出现时，在设置了某些寄存器后，会根据异常类型跳转到相应的固定地址处（高优先级异常服务入口），操作系统会将相应的异常处理程序，置于这些地址处。

龙芯 2E 的高优先级异常有：冷启动、热重启、非屏蔽中断，TLB 重填（32 位模式），xTLB 重填（64 位模式），cache 错误，其他异常。

龙芯 2E 之高优先级异常入口地址有以下五个：

Table A.1 优先级异常入口

异常类型	正常运行（**BEV 为 0)**	启动（**BEV 为 1)**
冷启动、热重启、非屏蔽中断	0xFFFFFFFF BFC00000	0xFFFFFFFF BFC00000
TLB 重填	0xFFFFFFFF 80000000	0xFFFFFFFF BFC00200
xTLB 重填	0xFFFFFFFF 80000080	0xFFFFFFFF BFC00280
cache 错误	0xFFFFFFFF A0000100	0xFFFFFFFF BFC00300
其他	0xFFFFFFFF 80000180	0xFFFFFFFF BFC00380

当龙芯正常运行时，STATUS 寄存器之 BEV 位为 0 ，0xFFFFFFFF 80000000 地址处不经 TLB 映射、但缓存；当龙芯启动时，STATUS 寄存器之 BEV 位为 1，0xFFFF FFFF BFC0 0200 地址处龙芯不缓存、不经 TLB 映射。MIPS 下 TLB、Cache 都要 OS 参与管理，在其启动时 OS 尚未接管系统，这个时候不采用 TLB、Cache 机制是很重要的。

注意，冷启动、热重启、非屏蔽中断的入口地址始终位于 0xFFFF FFFF BFC0 0000

由此可见龙芯 2E 启动时（冷启动异常或者热重启异常），执行的第一条指令是位于地址

0xFFFF FFFF BFC0 0000 处的，实际上龙芯电脑上所用之 BOOTLOADER(PMON)的第一条指令就映射在地址空间的 0xFFFF FFFF BFC0 0000 处。

注意到上面的五个固定的异常入口地址，有一个其他异常类入口（一般称为通用异常入口），当 CPU 内部异常或者外部中断发生时，CPU 硬件设置 CAUSE 寄存器的 ExcCode(CAUSE 6:2)位后，就跳转到该异常入口。ExcCode 位段用来描述通用异常类型，共 5 位，故而可以描述 2^5 = 32 个异常类型。

位于通用异常入口处的是操作系统设置的一个简单的异常处理程序，它会取出 CAUSE 寄存器的 ExcCode 域（5 位，可以描述 32 个异常），用之索引一个通用异常处理表(exception_handlers)，并跳转到异常处理表项所指向的处理程序，通用异常处理表如下所示：

```
-------------------------
|     handle_int        |     ---------->   中断          ExcCode 值为 0
-------------------------
|     handle_tlbm       |     ---------->   TLB 修改异常   ExcCode 值为 1
-------------------------
|     handle_tlbl       |     ---------->   TLB 读异常     ExcCode 值为 2
-------------------------
|       .......         |
|                       |
-------------------------
|     handle_sys        |     ---------->   系统调用       ExcCode 值为 8
-------------------------
|       .......         |
```

该异常处理表很像 x86 的 IDT，只是他的每个表项没什么附加信息，就是一个相应异常处理程序的地址。

来自硬件的中断，CPU 会自动将 CAUSE 寄存器的 ExcCode 域(6:2)设为 0，其最终会执行总的中断处理程序 handle_int。

ExcCode 位为 0 时，只是笼统地描述为中断，具体的是何种中断，还要借助 CAUSE 寄存器的 IP 位(15:8, IP7-IP0)来描述。硬件中断出现时，CPU 会根据中断信号的来源，设置

CAUSE 之 IP 位。IP 位共 8 位，每位对应一个中断。龙芯 2E 下，8 个中断的用途分配如下：

Table A.2 CAUSE 之 IP 位对应中断

IP0	软件中断	保留未用
IP1	软件中断	保留未用
IP2	硬件中断	北桥中断控制器中断
IP3	硬件中断	保留
IP4	硬件中断	保留
IP5	硬件中断	I8259A 中断
IP6	硬件中断	Perfcounter 溢出
IP7	时钟中断	Timer 中断

IP6, IP7 都是 CPU 内部产生。

可以看到，一个硬件中断的流程应该是这样的（以键盘为例）：

1. 用户击键后，键盘控制器 8042 产生中断，通过 I8259A 在 CPU 的中断引脚上，引起异常

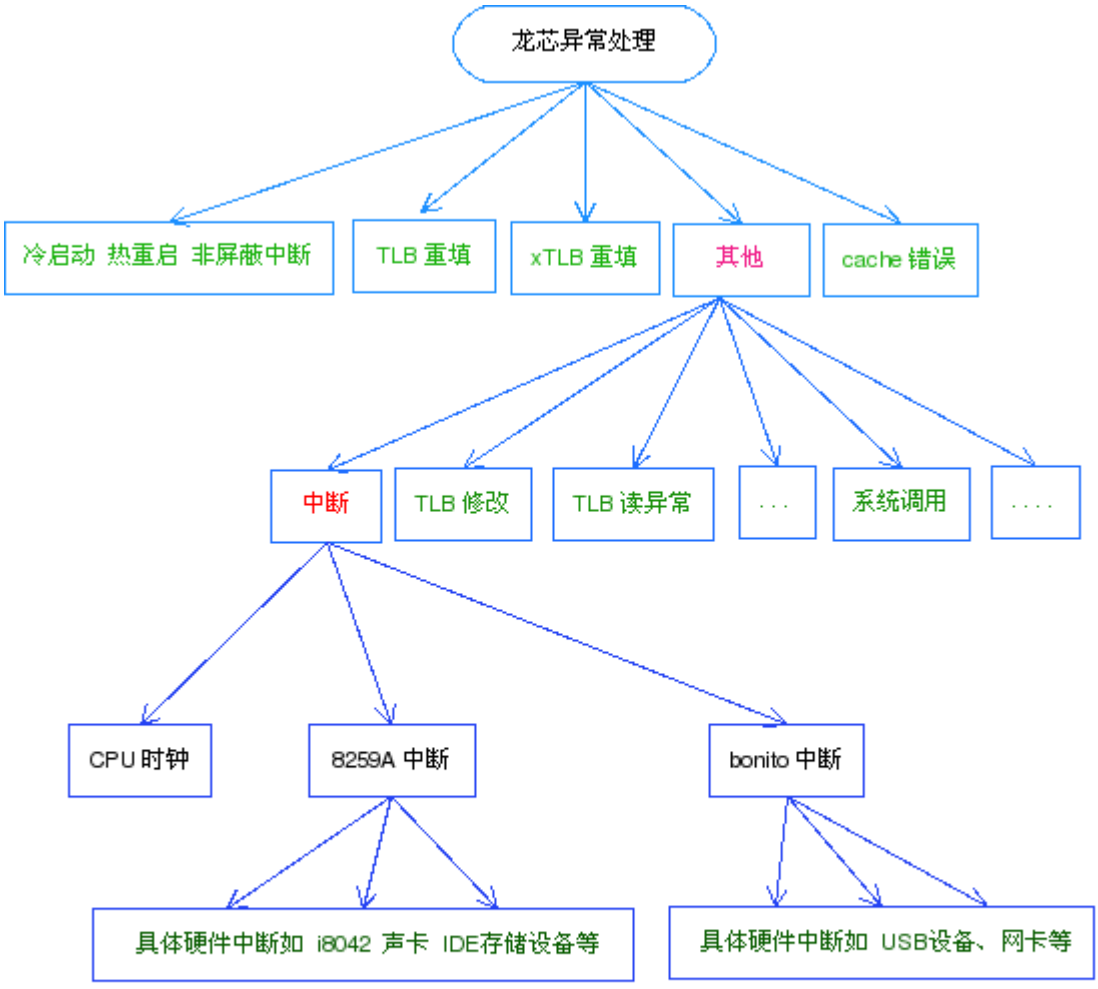
2. CPU 自动设置 CAUSE 的 ExcCode 位为 0，IP5 为 1，并跳转到通用异常入口 0xFFFFFFFF 80000180

3. 位于通用异常入口处的简单异常处理程序，根据 ExcCode 的值索引异常处理表 (exception_handlers)，获取到 0 号异常的处理程序是 handle_int，并跳转过去

4. handle_int 根据 CAUSE 之 IP 位的值跳转到中断控制器 8259A 相关的中断处理函数 do_nb2005_8259

5. do_nb2005_8259 读取 8259A 之 In-Service Register(ISR, 注意与 x86 的差异，x86 是由 8259A 主动将中断号送上数据总线的)，通过简单的计算得到中断号，进而调用 do_IRQ 进入相应的中断处理程序。

可以看到龙芯的整个异常处理是个树状结构，如下图所示：



龙芯II 异常处理结构

B. 内核代码分析

B.1 高优先级异常入口初始化

通用异常入口初始化，位于：

```
[arch/mips/kernel/traps.c]
void __init trap_init()
{
   ...
   set_handler(0x180, &except_vec3_generic, 0x80);
   ...
}
```

set_handler 亦定义于该文件中：

```
void __init set_handler (unsigned long offset, void *addr,
unsigned long size)
{
    memcpy((void *)(ebase + offset), addr, size);
    flush_icache_range(ebase + offset, ebase + offset + size);
}
```

很显然 set_handler 完成的操作是将 addr 处，大小为 size 字节的数据块复制到指定的异常入口处。其第一个参数指定入口的偏移地址，如 xTLB 重填为 0x100，通用异常为 0x180；第二个参数为异常处理程序的指针；第三个为需要复制的大小。其中 ebase 是个全局变量，32 位内核下其值为 0x8000 0000。特别地，每个高优先级异常处理程序的长度不能超过 128 bytes。下面看看复制到 0x8000 0180 处的是什么：

except_vec3_generic 是通用异常处理程序，定义于：

```
[arch/mips/kernel/genex.S]
/*
 * General exception vector for all other CPUs.
```

```
 *
 * Be careful when changing this, it has to be at most 128 bytes
 * to fit into space reserved for the exception handler.
 */
NESTED(except_vec3_generic, 0, sp)
    .set    push
    .set    noat
    mfc0    k1, CP0_CAUSE
    andi    k1, k1, 0x7c
#ifdef CONFIG_64BIT
    dsll    k1, k1, 1
#endif
    PTR_L   k0, exception_handlers(k1)
    jr  k0
    .set    pop
    END(except_vec3_generic)
```

这段程序完成的功能为：取 cause 寄存器之 ExcCode 值，然后跳转到 exception_handlers+ExcCode*4 处，注意 ExcCode 为 cause 寄存器的位 6:2，因此 CAUSE & 0x7c 就是 ExcCode*4。64 位下，指针长度为 8 Bytes，异常表的每项亦为 8 Bytes，则索引值为 ExcCode*8，因此要在原 32 位的基础上再左移一位。

cache 错误入口初始化，位于：

```
[arch/mips/mm/c-r4k.c]
void __init r4k_cache_init()
{
   ...
   set_uncached_handler(0x100, &except_vec2_generic, 0x80);
   ...
}
```

因为 cache 错误时可以 cache 的 KSEG1 段不能用了，则 cache 错误异常处理程序位于 KSEG1 之 0xA0000000 + 0x100 处，长度最大为 128 Bytes，异常处理程序为 except_vec2_generic，定义于：

```
[arch/mips/mm/cex-gen.S]
    LEAF(except_vec2_generic)
    .set    noreorder
    .set    noat
    .set    mips0
    /*
     * This is a very bad place to be.  Our cache error
     * detection has triggered.  If we have write-back data
     * in the cache, we may not be able to recover.  As a
     * first-order desperate measure, turn off KSEG0 cacheing.
     */
    mfc0    k0,CP0_CONFIG
    li  k1,~CONF_CM_CMASK
    and k0,k0,k1
    ori k0,k0,CONF_CM_UNCACHED
    mtc0    k0,CP0_CONFIG
    /* Give it a few cycles to sink in... */
    nop
    nop
    nop

    j   cache_parity_error
    nop
    END(except_vec2_generic)
```

另外 set_uncached_handler 定义于：

```
[arch/mips/kernel/traps.c]
```

```
void __init set_uncached_handler (unsigned long offset, void
*addr, unsigned long size)
{
#ifdef CONFIG_32BIT
    unsigned long uncached_ebase = KSEG1ADDR(ebase);
#endif
#ifdef CONFIG_64BIT
    unsigned long uncached_ebase = TO_UNCAC(ebase);
#endif

    memcpy((void *)(uncached_ebase + offset), addr, size);
}
```

其中 KSEG1ADDR 的宏用于将虚址 ebase 转化为对应的 uncached 段的虚址（32bit 下为 KSEG1）。32bit 下 ebase 为 0x8000 0000 则 KSEG1ADDR(ebase) 的值为 0xA000 0000。

以上是第一层的异常处理，可以看到每个异常处理的程序所完成的功能都比较精简，且长度都限制在 128 Bytes，32 条指令以内。其中频繁使用的 tlb refill 异常处理程序，不像上述的那样，事先编写好的，而是通过一些函数动态生成，然后复制到对应的入口处的。比如龙芯 2E 的 tlb_refill_handler 就是由位于 arch/mips/mm/tlbex.c 的 build_r4000_tlb_refill_handler 函数则初始化阶段生成的。至于为何采取这种方式，主要是因为要根据用户的配置生成适合各种 MIPS 平台的 tlb_refill_handler ，由于要考虑的情况过多，使用通常的条件编译的方式已经不能满足需求。

B.2 通用异常处理表初始化

在上面的通用异常处理程序的分析中我们看到，其用 CAUSE 寄存器的 ExcCode 索引一张通用异常处理表 exception_handlers，它定义于：

```
[arch/mips/kernel/traps.c]
unsigned long exception_handlers[32];
```

实际上是一个数组，每个元素是相应异常的处理程序地址。初始化时则函数 trap_init()中完成：

```
[arch/mips/kernel/traps.c]
void __init trap_init(void)
{
    ......
    /*
     * Setup default vectors
     */
    for (i = 0; i <= 31; i++)
        set_except_vector(i, handle_reserved);
    ......
    set_except_vector(0, handle_int);
    set_except_vector(1, handle_tlbm);
    set_except_vector(2, handle_tlbl);
    set_except_vector(3, handle_tlbs);

    set_except_vector(4, handle_adel);
    set_except_vector(5, handle_ades);

    set_except_vector(6, handle_ibe);
```

```
    set_except_vector(7, handle_dbe);

    set_except_vector(8, handle_sys);
    set_except_vector(9, handle_bp);
    set_except_vector(10, handle_ri);
    set_except_vector(11, handle_cpu);
    set_except_vector(12, handle_ov);
    set_except_vector(13, handle_tr);
    ......
    if (cpu_has_fpu && !cpu_has_nofpuex)
        set_except_vector(15, handle_fpe);

    set_except_vector(22, handle_mdmx);

    if (cpu_has_mcheck)
        set_except_vector(24, handle_mcheck);

    if (cpu_has_mipsmt)
        set_except_vector(25, handle_mt);

    if (cpu_has_dsp)
        set_except_vector(26, handle_dsp);
    ......
}
```

用于填充 exception_handlers 的 set_except_vector 定义于同一个文件中：

```
void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long) addr;
```

```
    unsigned long old_handler = exception_handlers[n];

    exception_handlers[n] = handler;
    if (n == 0 && cpu_has_divec) {
        *(volatile u32 *)(ebase + 0x200) = 0x08000000 |
                                           (0x03ffffff & (handler >>
2));
        flush_icache_range(ebase + 0x200, ebase + 0x204);
    }
    return (void *)old_handler;
}
```

函数所完成的主要操作即是将传来的异常处理函数的地址 addr 赋值给 exception_handlers 的元素 n。当 n 为 0 以及 CPU 具有除法异常时，要进行一些处理，这个与龙芯 2E 无关，我们不关心，略过。

在上面我们看到，内核首先用 handle_reserved 填充整个 exception_handlers，然后依次填充 0~13 号异常的处理函数，尔后则根据各 MIPS CPU 的特点填充相应的处理函数，如具有 FPU 且具有 FPU 异常的 MIPS CPU 则填充元素 15 为 handle_fpe。龙芯 2E 的 32 个异常号(CAUSE 之 ExcCode)之含义为：

Table A.3 CAUSE 之 ExcCode 位

ExcCode	Mnemonic	Description
0	**Int**	中断
1	Mod	TLB 修改异常
2	TLBL	TLB 异常（读或者取指令）
3	TLBS	TLB 异常（存储）
4	**AdEL**	地址错误异常（读或者取指令）
5	**AdES**	地址错误异常（存储）
6	IBE	总线错误异常（取指令）
7	DBE	总线错误异常（读或存储）
8	**Sys**	系统调用异常
9	Bp	断点异常
10	RI	保留指令异常
11	CpU	协处理器不可用异常
12	Ov	算术溢出异常
13	Tr	陷阱异常
14		保留
15	FPE	浮点异常
16~22		保留
23	WATCH	
24~30		保留
31		保留

特别地，注意 `handle_int` 往往与具体的硬件平台相关，有的可能在具体平台的初始化代码里重新填充 `exception_handlers[0]`。比如龙芯 2E 的福珑 `mini PC` 上就在 `arch/mips/godson/godson2e/irq.c` 里调用 `set_except_vector(0, godson2e_handle_int);`重新设置为 `godson2e_handle_int`。

1~3 号的异常处理函数 `handle_tlbm, handle_tlbl, handle_tlbs`，与

tlb_refill_handler 相同，亦是在 arch/mips/mm/tlbex.c 中分别由 build_r4000_tlb_modify_handler(), build_r4000_tlb_load_handler(), build_r4000_tlb_store_handler() 生成。

其他的则由宏 BUILD_HANDLER 生成：

[arch/mips/kernel/genex.S]

```
   356      BUILD_HANDLER adel ade ade silent       /* #4  */
   357      BUILD_HANDLER ades ade ade silent       /* #5  */
   358      BUILD_HANDLER ibe be cli silent         /* #6  */
   359      BUILD_HANDLER dbe be cli silent         /* #7  */
   360      BUILD_HANDLER bp bp sti silent          /* #9  */
   361      BUILD_HANDLER ri ri sti silent          /* #10 */
   362      BUILD_HANDLER cpu cpu sti silent        /* #11 */
   363      BUILD_HANDLER ov ov sti silent          /* #12 */
   364      BUILD_HANDLER tr tr sti silent          /* #13 */
   365      BUILD_HANDLER fpe fpe fpe silent        /* #15 */
   366      BUILD_HANDLER mdmx mdmx sti silent      /* #22 */
   367      BUILD_HANDLER watch watch sti verbose       /* #23 */
   368      BUILD_HANDLER mcheck mcheck cli verbose     /* #24 */
   369      BUILD_HANDLER mt mt sti verbose         /* #25 */
   370      BUILD_HANDLER dsp dsp sti silent        /* #26 */
   371      BUILD_HANDLER reserved reserved sti verbose /* others
*/
```

特别地，8 号异常（用于系统调用）没有使用 BUILD_HANDLER 的方式生成。

宏 BUILD_HANDLER 定义于同一文件中：

```
   352      .macro  BUILD_HANDLER exception handler clear verbose
   353      __BUILD_HANDLER \exception \handler \clear \verbose
_int
```

```
   354      .endm
```

宏 __BUILD_HANDLER 亦定义于同一文件中：

```
   337      .macro  __BUILD_HANDLER exception handler clear
verbose ext
   338      .align  5
   339      NESTED(handle_\exception, PT_SIZE, sp)
   340      .set    noat
   341      SAVE_ALL
   342      FEXPORT(handle_\exception\ext)
   343      __BUILD_clear_\clear
   344      .set    at
   345      __BUILD_\verbose \exception
   346      move    a0, sp
   347      jal do_\handler
   348      j   ret_from_exception
   349      END(handle_\exception)
   350      .endm
   351
```

NESTED 定义于：

```
[include/asm-mips/asm.h]
/*
 * NESTED - declare nested routine entry point
 */
#define NESTED(symbol, framesize, rpc)                  \
        .globl  symbol;                         \
        .align  2;                              \
```

```
        .type   symbol,@function;              \
        .ent    symbol,0;                      \
symbol:     .frame  sp, framesize, rpc
```

该宏用于产生一个函数头定义，伪操作 .frame 的给出，使其能进行函数调用。

宏 SAVE_ALL 用于保存上下文，即保存几乎所有的寄存器。这个我们则系统调用里详细讨论。

FEXPORT 定义于：

[include/asm-mips/asm.h]

```
/*
 * FEXPORT - export definition of a function symbol
 */
#define FEXPORT(symbol)                 \
        .globl  symbol;             \
        .type   symbol,@function;       \
symbol:
```

该宏用于导出函数符号 symbol，可以看到前面在 NESTED 中已经用 .globl 和.type 定义了一个函数符号 handle_\exception，现在又用之定义了一个函数符号 handle_\exception\ext，其中\ext 在调用时__BUILD_HANDLER 始终为 _int，这个\exception 则为 BUILD_HANDLER 传递的第一个参数： adel，ades 等。实际上这里定义了 2 个函数符号，以 adel 为例，一为 handle_ades，一为 handle_ades_int。

接着 343 行又使用了一个宏 __BUILD_clear_\clear 这个宏根据__BUILD_HANDLER 的第 3 个参数的不同有不同形式。从 356~371 行的所有 BUILD_HANDLER 的调用中我们发现第 3 个参数共有 4 种，分别为 sti，cli，fpe，ade。则：

```
  291       .macro  __build_clear_sti
```

```
292       STI                             #开启中断
293       .endm
294
295       .macro  __build_clear_cli
296       CLI                             #关闭中断
297       .endm
298
299       .macro  __build_clear_fpe
300       cfc1    a1, fcr31               #取浮点控制寄存器的值于 a1
301       li  a2, ~(0x3f << 12)           #生成 fcr31 的 MASK 值
302       and a2, a1                      #清 fcr31 的 17~12 位
303       ctc1    a2, fcr31               #重新写入 fcr31
304       STI                             #开启中断
305       .endm
306
307       .macro  __build_clear_ade
308       MFC0    t0, CP0_BADVADDR        #取 BadVAddr 的值
309       PTR_S   t0, PT_BVADDR(sp)       #写入参数栈的 BadVAddr 位置
310       KMODE                           #设置 cause 的位，进入内核模式
311       .endm
```

可以看到用的较多的 __build_clear_cli/sti，实际上被另两个宏 CLI/STI 所代替，其分别用于禁止中断和开启中断。其中宏 CLI, STI, KMODE 都定义于 include/asm-mips/stackframe.h。

```
        .macro  CLI
#if !defined(CONFIG_MIPS_MT_SMTC)
        mfc0    t0, CP0_STATUS          # 取 status 的值
        li  t1, ST0_CU0 | 0x1f          # 0x1000001f 入 t1
        or  t0, t1
```

```
        xori    t0, 0x1f            # 低五位置 0，其他位不变
        mtc0    t0, CP0_STATUS      # 写入 status
#else /* CONFIG_MIPS_MT_SMTC */
        /*
         * For SMTC, we need to set privilege
         * and disable interrupts only for the
         * current TC, using the TCStatus register.
         */
        mfc0    t0,CP0_TCSTATUS
     /* Fortunately CU 0 is in the same place in both registers */
     /* Set TCU0, TMX, TKSU (for later inversion) and IXMT */
        li  t1, ST0_CU0 | 0x08001c00
        or  t0,t1
        /* Clear TKSU, leave IXMT */
        xori    t0, 0x00001800
        mtc0    t0, CP0_TCSTATUS
        ehb
     /* We need to leave the global IE bit set, but clear EXL...*/
        mfc0    t0, CP0_STATUS
        ori t0, ST0_EXL | ST0_ERL
        xori    t0, ST0_EXL | ST0_ERL
        mtc0    t0, CP0_STATUS
#endif /* CONFIG_MIPS_MT_SMTC */
        irq_disable_hazard
        .endm
```

我们不关心 SMTC 的情况，则通常的就是绿色部分。所完成的操作即：将 status 寄存器的 CU 域置为 0x1，表示 CP0 可用；低五位（KSU ERL EXL IE)都置为 0，则进入内核模式(KSU=00)，禁止中断(IE=0)。最后的 irq_disable_hazard 则是一个宏，定义于 include/asm-mips/hazards.h，旨在根据不同的 MIPS 处理器，使用不同的消除

CP0 冒险指令填充之。有些 CPU 如 MIPS 10000 已经在硬件层面解决了 CP0 冒险，irq_disable_hazard 则是一个空宏。

```
        .macro  STI
#if !defined(CONFIG_MIPS_MT_SMTC)
        mfc0    t0, CP0_STATUS      # 取 status 的值
        li  t1, ST0_CU0 | 0x1f      # 0x1000001f 入 t1
        or  t0, t1
        xori    t0, 0x1e            # 位 5~1 置 0，位 0 置 1，其他位不变
        mtc0    t0, CP0_STATUS      # 写入 status
#else /* CONFIG_MIPS_MT_SMTC */
        /*
         * For SMTC, we need to set privilege
         * and enable interrupts only for the
         * current TC, using the TCStatus register.
         */
        ehb
        mfc0    t0,CP0_TCSTATUS
    /* Fortunately CU 0 is in the same place in both registers */
    /* Set TCU0, TKSU (for later inversion) and IXMT */
        li  t1, ST0_CU0 | 0x08001c00
        or  t0,t1
        /* Clear TKSU *and* IXMT */
        xori    t0, 0x00001c00
        mtc0    t0, CP0_TCSTATUS
        ehb
    /* We need to leave the global IE bit set, but clear EXL...*/
        mfc0    t0, CP0_STATUS
        ori t0, ST0_EXL
        xori    t0, ST0_EXL
```

```
        mtc0    t0, CP0_STATUS
     /* irq_enable_hazard below should expand to EHB for 24K/34K
cpus */
#endif /* CONFIG_MIPS_MT_SMTC */
        irq_enable_hazard
        .endm
```

只关心非 SMTC 部分，与 CLI 类似，所不同的是只是将位 5~1 置为了 0，位 0 置为 1。即：内核模式(KSU=00)，开启中断(IE=1)。

```
        .macro  KMODE
#ifdef CONFIG_MIPS_MT_SMTC
        /*
         * This gets baroque in SMTC.  We want to
         * protect the non-atomic clearing of EXL
         * with DMT/EMT, but we don't want to take
         * an interrupt while DMT is still in effect.
         */

        /* KMODE gets invoked from both reorder and noreorder code
*/
        .set    push
        .set    mips32r2
        .set    noreorder
        mfc0    v0, CP0_TCSTATUS
        andi    v1, v0, TCSTATUS_IXMT
        ori v0, TCSTATUS_IXMT
        mtc0    v0, CP0_TCSTATUS
        ehb
        DMT 2                   # dmt    v0
        /*
```

```
         * We don't know a priori if ra is "live"
         */
        move    t0, ra
        jal mips_ihb
        nop /* delay slot */
        move    ra, t0
#endif /* CONFIG_MIPS_MT_SMTC */
        mfc0    t0, CP0_STATUS
        li  t1, ST0_CU0 | 0x1e      # 0x1000001e 入t1
        or  t0, t1
        xori    t0, 0x1e
        mtc0    t0, CP0_STATUS
#ifdef CONFIG_MIPS_MT_SMTC
        ehb
        andi    v0, v0, VPECONTROL_TE
        beqz    v0, 2f
        nop /* delay slot */
        emt
2:
        mfc0    v0, CP0_TCSTATUS
        /* Clear IXMT, then OR in previous value */
        ori v0, TCSTATUS_IXMT
        xori    v0, TCSTATUS_IXMT
        or  v0, v1, v0
        mtc0    v0, CP0_TCSTATUS
        /*
         * irq_disable_hazard below should expand to EHB
         * on 24K/34K CPUS
         */
        .set pop
#endif /* CONFIG_MIPS_MT_SMTC */
```

```
        irq_disable_hazard
        .endm
```

依然只关心非 SMTC 部分，与 STI 类似，所不同的只是将 位 5~1 置为了 0，其他位不变。即：内核模式(KSU=00)。

接着看__BUILD_HANDLER 的下面，345 行是一个辅助宏，其根据第 4 个参数的值判断是否打印一些信息，其有种选择 verbose 和 silent。为 verbose 则是一空宏：

```
313     .macro  __BUILD_silent exception
314     .endm
```

为 verbose 则为：

```
320     .macro  __BUILD_verbose nexception
321     LONG_L  a1, PT_EPC(sp)
322 #ifdef CONFIG_32BIT
323     PRINT("Got \nexception at %08lx\012")
324 #endif
325 #ifdef CONFIG_64BIT
326     PRINT("Got \nexception at %016lx\012")
327 #endif
328     .endm
```

__BUILD_HANDLER 的 346~348 行，则为核心部分，前面的一些宏都是做了一些准备工作，到这里则将 sp 的值作为调用 do_\exception 的第一个参数，然后 jal 调用 do_\exception，返回后直接跳转到 ret_from_exception 进行一些扫尾工作。因此真正的功能实现是在 do_\exception 里，如地址错误的 adel, ades 都为 do_ade，其定义于 arch/mips/kernel/unaligned.c

END 也定义于同一个文件中：

```
/*
 * END - mark end of function
 */
#define END(function)                   \
        .end    function;               \
        .size   function,.-function
```

可以看到，文件 arch/mips/kernel/genex.S 的主要作用在于生成一系列的异常处理函数。包括高优先级的 except_vec3_generic，通用异常的 handle_ades, handle_adel 等等。

下面我们着重看看与外部设备密切相关的 0 号中断异常处理函数 handle_int，与系统调用相关的 8 号异常处理函数 handle_sys。

B.3 中断异常处理

负责中断异常处理的是 handle_int，其与具体的硬件平台紧密相关，有的可能在具体平台的初始化代码里重新填充 exception_handlers[0]。例如龙芯 2E 的福珑 mini PC 上就在 arch/mips/godson/godson2e/irq.c 里调用 set_except_vector(0, godson2e_handle_int)，重新设置其为 godson2e_handle_int。

```
[arch/mips/godson/godson2e/irq.c]
void __init arch_init_irq(void)
{
......
    /* Sets the first-level interrupt dispatcher. */
    set_except_vector(0, godson2e_handle_int);
......
}
```

其中 godson2e_handle_int 定义于：

```
[arch/mips/godson/godson2e/int-handler.S]

/*
 * godson2e_handle_int:
 * We check for the timer first, then check PCI ints A and D.
 * Then check for serial IRQ and fall through.
 */
        .set    noreorder
        .align  5
        NESTED(godson2e_handle_int, PT_SIZE, sp)
        .set    noat
```

```
        SAVE_ALL                        # 保存上下文
        CLI                             # 关中断
        .set    at

        /*
         * Get pending interrupts
         */
        mfc0    t0,CP0_CAUSE            # get pending interrupts
        mfc0    t2,CP0_STATUS           # get enabled interrupts
        and     t0,t2                   # isolate allowed ones

        andi    t1, t0, STATUSF_IP7     /* cpu timer */
        bnez    t1, ll_cputimer_irq

#ifdef CONFIG_PERFCTR
        andi    t1, t0, STATUSF_IP6     /* cpu performance ov*/
        bnez    t1, ll_perfctr_irq
#endif

        andi    t1, t0, STATUSF_IP5     /* int3 hardware line */
        bnez    t1, ll_8259_irq

        andi    t1, t0, STATUSF_IP2     /*int0 hardware line*/
        bnez    t1, ll_nb2005_irq

        .set    reorder
ll_spurious_irq:
        /* wrong alarm or masked ... */
        j   spurious_interrupt
        nop
        END(godson2e_handle_int)
```

```
        .align  5

ll_cputimer_irq:
        li  a0, 63
        move    a1, sp
        jal do_IRQ
        j   ret_from_irq
#ifdef CONFIG_PERFCTR
ll_perfctr_irq:
        li  a0, 62
        move    a1, sp
        jal do_IRQ
        j   ret_from_irq
#endif

ll_8259_irq:
        li  t0, 0xbfd00000      # 0xbfd00000 为 I/O 的基地址
        lb  t2, 0x20(t0)        # 读 0x20 端口
        lb  t3, 0xa0(t0)        # 读 0xa0 端口
        sll t3, t3, 8
        or  t2, t2, t3          # 合并
        move    a0, t2
        move    a1, sp
        and a0, 0xfffffffb;     # 剔除主片用于级联另一片的 IR2 信号
        jal do_nb2005_8259
        nop
        j   ret_from_irq        # 中断返回清理现场

ll_nb2005_irq:
        move    a0, sp
```

```
        jal bonito_irqdispatch
        j   ret_from_irq
```

godson2e_handle_int 是龙芯平台中断处理的骨架函数。其首先使用 SAVE_ALL 宏，保存上下文，这个宏定义于：

[include/asm-mips/stackframe.h]

```
        .macro  SAVE_ALL
        SAVE_SOME
        SAVE_AT
        SAVE_TEMP
        SAVE_STATIC
        .endm
```

其又由四个宏组成，都定义于同一文件中：

```
        .macro  SAVE_SOME
        .set    push
        .set    noat
        .set    reorder
        mfc0    k0, CP0_STATUS
        sll     k0, 3           /* extract cu0 bit */
        .set    noreorder
        bltz    k0, 8f          # k0 小于 0（最高位为 1）则为内核模式
         move   k1, sp
        .set    reorder
        /* Called from user mode, new stack. */
        get_saved_sp
8:      move    k0, sp
        PTR_SUBU sp, k1, PT_SIZE  # PT_SIZE=sizeof(struct pt_regs)
        LONG_S  k0, PT_R29(sp)    # 32bit LONG_S = sw
```

```
        LONG_S  $3, PT_R3(sp)       # 64bit LONG_S = sd
        /*
         * You might think that you don't need to save $0,
         * but the FPU emulator and gdb remote debug stub
         * need it to operate correctly
         */
        LONG_S  $0, PT_R0(sp)
        mfc0    v1, CP0_STATUS
        LONG_S  $2, PT_R2(sp)
        LONG_S  v1, PT_STATUS(sp)
#ifdef CONFIG_MIPS_MT_SMTC
        /*
         * Ideally, these instructions would be shuffled in
         * to cover the pipeline delay.
         */
        .set    mips32
        mfc0    v1, CP0_TCSTATUS
        .set    mips0
        LONG_S  v1, PT_TCSTATUS(sp)
#endif /* CONFIG_MIPS_MT_SMTC */
        LONG_S  $4, PT_R4(sp)
        mfc0    v1, CP0_CAUSE
        LONG_S  $5, PT_R5(sp)
        LONG_S  v1, PT_CAUSE(sp)
        LONG_S  $6, PT_R6(sp)
        MFC0    v1, CP0_EPC
        LONG_S  $7, PT_R7(sp)
#ifdef CONFIG_64BIT
        LONG_S  $8, PT_R8(sp)
        LONG_S  $9, PT_R9(sp)
#endif
```

```
        LONG_S  v1, PT_EPC(sp)
        LONG_S  $25, PT_R25(sp)
        LONG_S  $28, PT_R28(sp)
        LONG_S  $31, PT_R31(sp)
        ori     $28, sp, _THREAD_MASK  #_THREAD_MASK = PAGE_SIZE-1
        xori    $28, _THREAD_MASK
        .set    pop
        .endm    /* SAVE_SOME end */
```

其中开始处宏 get_saved_sp:

```
        .macro  get_saved_sp    /* Uniprocessor variation */
#ifdef CONFIG_64BIT
        lui k1, %highest(kernelsp)
        daddiu  k1, %higher(kernelsp)
        dsll    k1, k1, 16
        daddiu  k1, %hi(kernelsp)
        dsll    k1, k1, 16
#else
        lui k1, %hi(kernelsp)
#endif
        LONG_L  k1, %lo(kernelsp)(k1)  # 将当前进程内核栈的地址放入 k1
        .endm
```

只看 32bit 部分，kernelsp 为一指针，存放着当前进程内核栈的地址。get_saved_sp 就是将当前进程内核栈的地址放入 k1。

回头看 SAVE_SOME 宏的开始部分（开头绿色部分），其根据 STATUS 左移 3 位的值，判断是否为用户模式，如果是，则要切换内核栈，否则直接跳转到标号 8 处。接下来则是在内核栈上保存 $0, $2, $3, $4~$7, $8~$9(64bit), $25, $28, $29, $31, STATUS, CAUSE, EPC 的值。保存好的值正好在内核栈上成一个 struct pt_regs 结构

```
        .macro  SAVE_AT
        .set    push
        .set    noat
        LONG_S  $1, PT_R1(sp)
        .set    pop
        .endm
```

宏 SAVE_AT 旨在保存 $1 的值于内核栈上。

```
        .macro  SAVE_TEMP
        mfhi    v1                  # 取 hi 寄存器的值入 v1
#ifdef CONFIG_32BIT
        LONG_S  $8, PT_R8(sp)
        LONG_S  $9, PT_R9(sp)
#endif
        LONG_S  v1, PT_HI(sp)
        mflo    v1
        LONG_S  $10, PT_R10(sp)
        LONG_S  $11, PT_R11(sp)
        LONG_S  v1,  PT_LO(sp)
        LONG_S  $12, PT_R12(sp)
        LONG_S  $13, PT_R13(sp)
        LONG_S  $14, PT_R14(sp)
        LONG_S  $15, PT_R15(sp)
        LONG_S  $24, PT_R24(sp)
        .endm
```

宏 SAVE_TEMP 旨在保存 $8~$9(32bit), $10~$15, $24, hi, lo，相较于 SAVE_SOME 其所保存的都是些不是非常重要的寄存器，对应于寄存器约定里的 temp 寄存器。

```
        .macro  SAVE_STATIC
        LONG_S  $16, PT_R16(sp)
        LONG_S  $17, PT_R17(sp)
        LONG_S  $18, PT_R18(sp)
        LONG_S  $19, PT_R19(sp)
        LONG_S  $20, PT_R20(sp)
        LONG_S  $21, PT_R21(sp)
        LONG_S  $22, PT_R22(sp)
        LONG_S  $23, PT_R23(sp)
        LONG_S  $30, PT_R30(sp)
        .endm
```

宏 SAVE_TEMP 旨在保存 $16~$23, $30 的值，对应于寄存器约定里的 static 寄存器

合起来，SAVE_ALL 这个宏旨在当前进程内核栈上保存上下文，整个保存的数据结构由定义于 include/asm-mips/ptrace.h 的 struct pt_regs 描述：

```
struct pt_regs {
#ifdef CONFIG_32BIT
    /* Pad bytes for argument save space on the stack. */
    unsigned long pad0[6];
#endif

    /* Saved main processor registers. */
    unsigned long regs[32];

    /* Saved special registers. */
    unsigned long cp0_status;
    unsigned long hi;
    unsigned long lo;
    unsigned long cp0_badvaddr;
    unsigned long cp0_cause;
```

```
    unsigned long cp0_epc;

#ifdef CONFIG_MIPS_MT_SMTC
    unsigned long cp0_tcstatus;
    unsigned long smtc_pad;
#endif /* CONFIG_MIPS_MT_SMTC */
};
```

各寄存器的偏移常量（如 PT_R10），由定义于 arch/mips/kernel/asm-offsets.c 的函数 output_ptreg_defines 动态生成。

接着 godson2e_handle_int 往下看，注意到 CAUSE 的 IP 位(15:8)与 STATUS 的中断屏蔽位 IM(15:8)都位于 15:8，因此直接相与即可得有效的 IP 位。参考 Table A.2，当硬件中断出现时，CPU 会根据中断信号的来源，置 CAUSE 寄存器的对应 IP 位为 1。因此在中断处理函数中，只要读取之，即可获得具体的中断来源。故而则保存了上下文，禁止了中断、切换到内核模式后，其测试 IP 位即可跳转到相应的处理函数中。

IP7 为 1 时，时钟中断，直接将中断通道号 63 作为第一个参数，栈指针作为第二个参数，调用 do_IRQ 即进入对应的中断通道处理函数。

IP6 为 1 时，性能计数器溢出，直接将中断通道号 62 作为第一个参数，栈指针作为第二个参数，调用 do_IRQ 即进入对应的中断通道处理函数。

IP5 为 1 时，中断信号来自于中断控制器 8259A，则首先分别读取两片 8259A 的 IRR 值，合并在一起，作为第一个参数传给定义于 arch/mips/godson/godson2e/irq.c 的 do_nb2005_8259，在这个函数中，一一测试是具体的 16 个中断中的哪个，然后将对应的中断通道号传给 do_IRQ。顺便说一句，此处直接读取 8259A 的 IRR 值，来判断具体的中断不是很合适。8259A 在没有 INTA 信号时，有一个轮询命令(poll command)，可以用之读取到 ISR 的值，这个可以直接使用定义于 include/asm-mips/i8259.h 的 i8259_irq 函数，获取 ISR 的值来判断具体的中断比较合适。（最新版本的内核已经修正为使用 i8259_irq 来获取具体中断）

IP2 为 1 时，中断信号来自于北桥内部中断控制器 bonito，则进入定义于 arch/mips/godson/godson2e/irq.c 的 bonito_irqdispatch，在其中读取该中断控制器的 ISR 值，即可确认具体的中断，然后将对应的中断通道号传给 do_IRQ。

关于中断通道号，目前龙芯 2E 平台上的分配是这样的：

0~15	对应 8259A 的 16 个中断（实际上有效的为 15，主片 IR2 用于级联）
16~47	对应北桥内部中断控制器的 32 个中断
56~63	对应 MIPS CPU 的 8 个中断，主要用的就是 62，63

中断处理的最后都要跳转到 ret_from_irq，这个符号定义于：

```
[arch/mips/kernel/entry.S]
FEXPORT(ret_from_exception)
    preempt_ stop
FEXPORT(ret_from_irq)
    LONG_L  t0, PT_STATUS(sp)        # returning to kernel mode?
    andi    t0, t0, KU_USER
    beqz    t0, resume_kernel        # 内核模式下则跳转

resume_userspace:                    # 用户模式
    local_irq_disable                # 关中断
    LONG_L  a2, TI_FLAGS($28)        # 获取当前线程的 Thread
                                       Information Flags 标志
    andi    t0, a2, _TIF_WORK_MASK   # 忽略不需考虑的情况（参见 include/
                                       asm-mips/thread_info.h）
    bnez    t0, work_pending         # 有情况要处理，则跳转
    j   restore_all

#ifdef CONFIG_PREEMPT
resume_kernel:
    local_irq_disable                # 关中断
```

```
    lw  t0, TI_PRE_COUNT($28)       # 当前线程抢占计数入 t0
    bnez    t0, restore_all         # 不为 0，就圆满
need_resched:                       # 为 0，则还需判断是否需要调度
    LONG_L  t0, TI_FLAGS($28)
    andi    t1, t0, _TIF_NEED_RESCHED
    beqz    t1, restore_all
    LONG_L  t0, PT_STATUS(sp)       # 判断异常处理前中断是否关闭
    andi    t0, 1
    beqz    t0, restore_all         # 如果关闭，则圆满
    jal preempt_schedule_irq        # 调度
    b   need_resched
#endif

work_pending:                       #a2 中已放入当前线程的 TI_FLAGS
    andi    t0, a2, _TIF_NEED_RESCHED   # 是否需要调度？
    beqz    t0, work_notifysig          # 不需要
work_resched:
    jal schedule                        # 调度

    local_irq_disable       # make sure need_resched and
                            # signals dont change between
                            # sampling and return
    LONG_L  a2, TI_FLAGS($28)
    andi    t0, a2, _TIF_WORK_MASK
    beqz    t0, restore_all
    andi    t0, a2, _TIF_NEED_RESCHED
    bnez    t0, work_resched

work_notifysig:             # deal with pending signals and
                            # notify-resume requests
    move    a0, sp
    li  a1, 0
```

```
    jal do_notify_resume        # a2 already loaded
    j   resume_userspace
```

do_notify_resume 定义于 arch/mips/kernel/signal.c，用于通知当前线程恢复用户空间执行。下面看看用于善后的 restore_all:

```
FEXPORT(restore_all)            # restore full frame
#ifdef CONFIG_MIPS_MT_SMTC
/* Detect and execute deferred IPI "interrupts" */
    move    a0,sp
    jal deferred_smtc_ipi
/* Re-arm any temporarily masked interrupts not explicitly "acked"
*/
    mfc0    v0, CP0_TCSTATUS
    ori v1, v0, TCSTATUS_IXMT
    mtc0    v1, CP0_TCSTATUS
    andi    v0, TCSTATUS_IXMT
    ehb
    mfc0    t0, CP0_TCCONTEXT
    DMT 9                   # dmt t1
    jal mips_ihb
    mfc0    t2, CP0_STATUS
    andi    t3, t0, 0xff00
    or  t2, t2, t3
    mtc0    t2, CP0_STATUS
    ehb
    andi    t1, t1, VPECONTROL_TE
    beqz    t1, 1f
    EMT
1:
    mfc0    v1, CP0_TCSTATUS
    /* We set IXMT above, XOR should clear it here */
```

```
    xori    v1, v1, TCSTATUS_IXMT
    or  v1, v0, v1
    mtc0    v1, CP0_TCSTATUS
    ehb
    xor t0, t0, t3
    mtc0    t0, CP0_TCCONTEXT
#endif /* CONFIG_MIPS_MT_SMTC */
    .set    noat
    RESTORE_TEMP
    RESTORE_AT
    RESTORE_STATIC
FEXPORT(restore_partial)        # restore partial frame
    RESTORE_SOME
    RESTORE_SP_AND_RET
    .set    at
```

依旧不关心 SMTC 的情形，RESTORE_TEMP, RESTORE_AT, RESTORE_STATIC, RESTORE_SOME 依次执行与保存相对应的恢复操作，最后的宏 RESTORE_SP_AND_RET 用于恢复栈、以及异常返回，定义于：

[include/asm-mips/stackframe.h]

```
        .macro  RESTORE_SP_AND_RET
        LONG_L  sp, PT_R29(sp)          # 恢复原来的栈指针
#       HIGH_L  sp, PT_HI_R29(sp) /* don't do this! sp has
changed. */
        .set    mips3
        eret                            # 异常处理返回
        .set    mips0
        .endm
```

eret 执行 2 个操作，一为清 status 之 EXL 位，二为跳转到 EPC 指向的地址处。

最后关心一下当一个中断出现时，MIPS CPU 所完成的动作：

1. 将 EPC 指向当前正在执行的指令，以便返回时恢复之
2. 置 STATUS 的 EXL 位为 1（其他位不改变）
3. 设置 CAUSE 的 ExcCode 位为 0，指明为中断
4. 跳转到高优先级异常入口

特别地，当 STATUS 的 EXL 置为 1 时，意味着：
I. CPU 自动进入内核模式（忽略 STATUS[KSU]位），禁止中断（忽略 STATUS[IE]位）
II. TLB refill 异常入口将使用通用异常入口而非原 tlb refill 异常入口
III. CPU 将不响应新异常（EPC 不会被重新设置）

因此，为了支持嵌套异常，应尽可能快的将 STATUS 的 EXL 清零（当然首先要保存 EPC 的值），同时将 STATUS[KSU]=00（内核模式）、STATUS[IE]＝0（禁止中断）。注意到上面的 godson2e_int_handle 中，首先 SAVE_ALL，然后再使用 CLI 宏来完成这个操作，干净漂亮：

```
mfc0    t0, CP0_STATUS      # 取 status 的值
li  t1, ST0_CU0 | 0x1f      # 0x1000001f 入 t1
or  t0, t1
xori    t0, 0x1f            # 低五位置 0，其他位不变
mtc0    t0, CP0_STATUS      # 写入 status
```

STATUS 中还有一个重要的位是 ERL(Error Level)，当冷启动、热重启、非屏蔽中断（Reset, Soft Reset, NMI）或者 Cache 错误（Cache error）异常出现时，CPU 会将其置为 1，则：

I. CPU 自动禁止中断（忽略 STATUS[IE]位）
II. eret 使用 ErrorEPC 代替 EPC 作为返回地址
III. Kuseg 和 xkuseg 被改为 unmaped, uncached 的区域，以便在 cache 错误时，内存空间能正常访问

B.4 系统调用异常处理

下面看看用于系统调用异常处理 handle_sys，不同的 ABI 所用之 handle_sys 不同，我们仅看 o32 的：

[arch/mips/kernel/scall32-o32.S]

```
28  NESTED(handle_sys, PT_SIZE, sp)
29      .set    noat
30      SAVE_SOME                   # 保存部分重要的寄存器，详见 P28 的分析
31      STI                         # 设置 KSU=00（内核模式），IE=1（开中断）
                                      详见 P20 的分析
32      .set    at
33
34      lw  t1, PT_EPC(sp)          # skip syscall on return
35
36  #if defined(CONFIG_BINFMT_IRIX)
37      sltiu   t0, v0, MAX_SYSCALL_NO + 1 # check syscall number
38  #else
39      subu    v0, v0, __NR_O32_Linux                  #系统调用号减去基号
40      sltiu   t0, v0, __NR_O32_Linux_syscalls + 1     #小于则置 t0 为 1
41  #endif
42      addiu   t1, 4               # skip to next instruction
43      sw  t1, PT_EPC(sp)
44      beqz    t0, illegal_syscall                     #非法则跳转
45
```

先来看看当 syscall 指令引起一个系统调用异常时，MIPS CPU 做了什么：

1. 将引起异常的 syscall 指令所在地址压入 EPC
2. 置 STATUS 的 EXL 位为 1（其他位不改变）
3. 设置 CAUSE 的 ExcCode 位为 8，指明为系统调用
4. 跳转到高优先级异常入口

则在系统调用函数服务完后，eret 返回前，应将 EPC 的值加 4，即返回 syscall 的下一条指令处，否则的话就不停的进系统调用异常了;) 34、42、43 处的操作就是做这个的。

36~41 行处，我们不关心 IRIX 的情形，则 39、40、44 行检查系统调用号是否超出范围，如果超出范围则跳转到 illegal_syscall 处处理之。

在 MIPS Linux 中系统调用的约定是这样的：

v0: 置系统调用号
a0~a3: 置前四个参数，后面的参数用栈传

常数 __NR_O32_Linux 定义于 include/asm-mips/unistd.h，表示 ABI O32 的系统调用号的起始号，目前为 4000； __NR_O32_Linux_syscalls 定义于同一文件中，表示 ABI O32 的系统调用个数，目前是 308。

```
[handle_sys]
    46      sll t0, v0, 3           # 生成系统调用表之索引项
    47      la  t1, sys_call_table  # 系统调用表的始地址入 t1
    48      addu    t1, t0          # 索引系统调用表
    49      lw  t2, (t1)            # 从表中读取系统调用函数地址入 t2
    50      lw  t3, 4(t1)           # 从表中第二项读取系统调用的参数个数
    51      beqz    t2, illegal_syscall # 函数地址为 0 则该项无效
    52
    53      sw  a3, PT_R26(sp)      # save a3 for syscall restarting
    54      bgez    t3, stackargs   # >= 0，参数个数大于 4，则要额外的处理
    55
```

sys_call_table 亦定义于该文件中：

```
EXPORT(sys_call_table)
    syscalltable
    .size   sys_call_table, . - sys_call_table
```

```
    .macro  syscalltable
#if defined(CONFIG_BINFMT_IRIX)
    mille   sys_ni_syscall      0   /*    0 -  999 SVR4 flavour */
    mille   sys_ni_syscall      0   /* 1000 - 1999 32-bit IRIX */
    mille   sys_ni_syscall      0   /* 2000 - 2999 BSD43 flavour */
    mille   sys_ni_syscall      0   /* 3000 - 3999 POSIX flavour */
#endif

    sys sys_syscall     8   /* 4000 */
    sys sys_exit        1
    sys sys_fork        0
    sys sys_read        3
    sys sys_write       3
    sys sys_open        3   /* 4005 */
    sys sys_close       1
    sys sys_waitpid     3
    sys sys_creat       2
    sys sys_link        2
    sys sys_unlink      1   /* 4010 */
    sys sys_execve      0
    sys sys_chdir       1
    ......
    ......
    sys sys_sync_file_range 7   /* 4305 */
    sys sys_tee         4
    sys sys_vmsplice        4
    sys sys_move_pages      6
    .endm
```

内核将 308 个系统调用组织成一张表 **sys_call_table**，共有 308 行，两列，第一列是系统调用函数地址（4 字节），第二列是系统调用参数个数（4 字节）。每行占用 8 字节，因此 46 行处要将原索引号上乘一个 8。

注意生成系统调用表的宏 **sys:**

```
.macro  sys function, nargs
PTR \function
LONG    (\nargs << 2) - (5 << 2)  #系统调用参数个数在 5~8 时，该项 >= 0
.endm
```

第二项只有在系统调用参数个数为 5~8（最大参数个数为 8）时该项值大于等于零。回到上面 50、54 行的操作，当参数个数大于 4 时需要到 stackargs 处去处理。因为 4 个以后的参数是通过栈传过来的，因此 stackargs 主要负责将位于用户空间的参数复制到内核空间（内核栈）。

如果参数个数小于等于 4 个，则继续下面的：

```
56  stack_done:
57      lw  t0, TI_FLAGS($28)    #获取当前线程的 Thread Information Flags 标志
58      li  t1, _TIF_SYSCALL_TRACE | _TIF_SYSCALL_AUDIT
59      and t0, t1               # 测试是否需要系统调用跟踪和审计
60      bnez    t0, syscall_trace_entry          #需要，则跳转
61
62      jalr    t2               # t2 的内容为系统调用函数地址，见 49 行
63
64      li  t0, -EMAXERRNO - 1   # error?
65      sltu    t0, t0, v0
66      sw  t0, PT_R7(sp)        # set error flag
67      beqz    t0, 1f           # 返回值正常则跳到 1: 处
68
69      negu    v0               # error
70      sw  v0, PT_R0(sp)        # set flag for syscall
71                               # restarting
72  1:  sw  v0, PT_R2(sp)        # result
73
```

64~72 行完成函数调用后的返回值设置。

```
74  o32_syscall_exit:
75      local_irq_disable        # 关中断
76
```

```
77
78      lw  a2, TI_FLAGS($28)    #获取当前线程的 Thread Information Flags 标志
79      li  t0, _TIF_ALLWORK_MASK
80      and t0, a2               #忽略不需考虑的情况（参见 include/
                                   asm-mips/thread_info.h）
81      bnez    t0, o32_syscall_exit_work        #有情况要处理则跳转
82
83      j   restore_partial      #上下文恢复，eret 返回。详见 P37 处分析
84
```

到此处一个 **4** 参数以内，当前线程没有 **TIF** 标志的系统调用圆满矣！

```
85  o32_syscall_exit_work:
86      j   syscall_exit_work_partial
89
```

syscall_exit_work_partial 定义于 arch/mips/kernel/entry.S:

```
FEXPORT(syscall_exit_work_partial)
    SAVE_STATIC
syscall_exit_work:
    li  t0, _TIF_SYSCALL_TRACE | _TIF_SYSCALL_AUDIT
    and t0, a2                  # a2 is preloaded with TI_FLAGS
    beqz    t0, work_pending    # 详见 P35 处
    local_irq_enable            # 关中断

    move    a0, sp
    li  a1, 1
    jal do_syscall_trace        # 系统调用跟踪，定义于 arch/mips/kernel/ptrace.c
    b   resume_userspace        # 详见 P34 处
```

其检查当前线程的 TIF，看是否需要系统调用跟踪和审计，如果不需要，则到 work_pending 处，检查是否需要调度，如果不需调度则直接发送信号通知用户空间进程恢复执行；如果需调度则调度之，完了再检查 TIF 直到需调度位消失，立即跳转到 restore_all 处，则又圆满矣！

若需要系统调用跟踪和审计，则调用 do_syscall_trace 跟踪系统调用，返回后直接到 resume_userspace 处，由 work_pending 负责搞定其所关心的 TIF 标志后，进入 restore_all，则又又圆满矣！！ :)

```
 90  syscall_trace_entry:
 91      SAVE_STATIC
 92      move    s0, t2
 93      move    a0, sp
 94      li  a1, 0
 95      jal do_syscall_trace
 96
 97      move    t0, s0
 98      RESTORE_STATIC
 99      lw  a0, PT_R4(sp)       # Restore argument registers
100      lw  a1, PT_R5(sp)
101      lw  a2, PT_R6(sp)
102      lw  a3, PT_R7(sp)
103      jalr    t0
104
105      li  t0, -EMAXERRNO - 1  # error?
106      sltu    t0, t0, v0
107      sw  t0, PT_R7(sp)       # set error flag
108      beqz    t0, 1f
109
110      negu    v0          # error
111      sw  v0, PT_R0(sp)       # set flag for syscall
112                      # restarting
113  1:  sw  v0, PT_R2(sp)       # result
114
115      j   syscall_exit
```

90~115 行这一段完成 syscall trace 前的准备，以及 syscall trace 后的善后（返回值写入）。我们不关心，略过。

下面重点来看看系统调用的参数个数大于 4 时，stackargs 的处理。

因为系统调用的参数个数不定，因此就需要判断参数个数为 5、6、7、8 不同情况时，相应的复制操作个数。5 个参数时需要复制个数为 1，6 个时为 2，以此类推。

通常的解决方法是用四条分支判断语句判断四种情况，这样的实现是对流水线很不友好的，看看高人们的实现：

```
124  stackargs:
125       lw   t0, PT_R29(sp)       # 用户空间栈指针入 t0
126
127       /*
128        * We intentionally keep the kernel stack a little below the top
of
129        * userspace so we don't have to do a slower byte accurate check
here.
130        */
131       lw       t5, TI_ADDR_LIMIT($28)    # 0 ~ thread_info->addr_limit
                                             # 表示 thread address space
132       addu     t4, t0, 32
133       and      t5, t4
134       bltz     t5, bad_stack             # -> sp is bad
```

$28 保存有指向当前线程的 thread_info 结构的指针，TI_ADDR_LIMIT 由 arch/mips/kernel/asm-offsets.c 动态生成，是为 thread_info 的成员 addr_limit 相对于指向 thread_info 指针的偏移。addr_limit 保存有 thread address space 的最大值：用户空间线程为 0xBFFFFFFF，内核线程为 0xFFFFFFFF。

如果用户空间栈指针加上 32 后，与上 thread_info->addr_limit 的最高位为 1（补码为负数）则用户空间栈指针越界，是为 bad_stack.

```
135
136       /* Ok, copy the args from the luser stack to the kernel stack.
137        * t3 is the precomputed number of instruction bytes needed to
```

```
138         * load or store arguments 6-8.
139         */
140
141        la   t1, 5f              # 标号 5 所示之地址入 t1
142        subu     t1, t3          # t3 的内容为当前系统调用参数个数减去 5，再乘以 4
                                    这个已经预先计算好，保存于系统调用表每项的第二个
                                    字段，用时直接载入。另外，此前已经判断过，t3 的
                                    内容大于等于 0
143    1:  lw   t5, 16(t0)          # 从用户空间复制第 5 个参数
                                    t0 的内容为用户空间第一个参数的地址

144        .set     push
145        .set     noreorder
146        .set     nomacro
147        jr   t1
148         addiu   t1, 6f - 5f   #妙用分支延迟槽
149
150    2:  lw   t8, 28(t0)          # argument #8 from usp
151    3:  lw   t7, 24(t0)          # argument #7 from usp
152    4:  lw   t6, 20(t0)          # argument #6 from usp
153    5:  jr   t1
154         sw t5, 16(sp)           # argument #5 to ksp
155
156    C:  sw   t8, 28(sp)          # argument #8 to ksp
157    B:  sw   t7, 24(sp)          # argument #7 to ksp
158    A:  sw   t6, 20(sp)          # argument #6 to ksp
159    6:  j    stack_done          # go back
160         nop
```

只用两条 jr 指令，效率大大提高！简直妙不可言！

(1) t3 的值，在参数个数为 5 时，t3 为 0，6 时为 4，7 时为 8，8 时为 12 这个 t3 在这里，实际上是用来表示相对于标号 5 处的地址偏移！ 因为 mips/godson 下，指令的长度都是 4 个字节。因此 t3 值为 4（参数个数为 6）时，第一个 jr t1 是跳转到标号 4 处开始执行的。

(2) 第一个 jr t1 用来解决从用户空间复制数据操作的个数问题，相对应的，则是解决写入操作的个数问题，这个用第二个 jr t1 来解决。在此之前，更新 t1 的指令(addiu)的位置放的很巧妙，置于第一个 jr t1 的延迟槽中，不占用标号 4 与标号 5 之间的空间。

(3) 可以看到参数个数为 5 时，第二个 jr t1 直接跳转到标号 6 处执行，将第 5 个参数写入内核栈的操作置于延迟槽中；参数个数为 6 时，会跳转到标号 A 处执行；参数个数为 8 时，会跳转到标号 C 处执行，依次完成第 5、8、7、6 参数的写入。

该段程序的作者对 MIPS 平台下的延迟槽有深刻的理解，故而才能有如此神乎其技的妙用。

以下是一些特殊情况的处理，我们不关心，略去。

```
   161      .set    pop
   163      .section __ex_table,"a"
   164      PTR 1b,bad_stack
   165      PTR 2b,bad_stack
   166      PTR 3b,bad_stack
   167      PTR 4b,bad_stack
   168      .previous
   169
   170      /*
   171       * The stackpointer for a call with more than 4 arguments is
bad.
   172       * We probably should handle this case a bit more drastic.
   173       */
   174  bad_stack:
   175      negu    v0              # error
   176      sw  v0, PT_R0(sp)
   177      sw  v0, PT_R2(sp)
   178      li  t0, 1               # set error flag
   179      sw  t0, PT_R7(sp)
   180      j   o32_syscall_exit
   181
```

```
182      /*
183       * The system call does not exist in this kernel
184       */
185  illegal_syscall:
186      li  v0, -ENOSYS         # error
187      sw  v0, PT_R2(sp)
188      li  t0, 1               # set error flag
189      sw  t0, PT_R7(sp)
190      j   o32_syscall_exit

191      END(handle_sys)
```

至此，系统调用异常处理函数 handle_sys 分析完毕矣，至于具体的系统调用如：sys_fork, sys_execve 等则在具体的文件中实现，如 sys_fork 就在 arch/mips/kernel/syscall.c 中实现：

```
save_static_function(sys_fork);
__attribute_used__ noinline static int
_sys_fork(nabi_no_regargs struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.regs[29], &regs, 0, NULL, NULL);
}
```

Appendix A

add a system call

添加一个系统调用有以下几个步骤：

1. 在 kernel 的系统调用表 sys_call_table 中添加你的系统调用入口

根据你用的内核位数(32/64bit) 选择需要修改的文件：

```
arch/mips/kernel/scall32-o32.S  ------> 32bit kernel
arch/mips/kernel/scall64-64.S    ------> 64bit kernel
```

64bit 内核，如果支持兼容 32 bit ABI (o32, n32) 则还需修改 scall64-o32.S 或 scall64-n32.S

以 32bit 内核为例，在 arch/mips/kernel/scall32-o32.S 中找到宏定义

```
  .macro  syscalltable            /* 系统调用号从 4000 开始 */
    ......
    ......
    sys sys_timerfd       4
    sys sys_eventfd       1
    sys sys_fallocate     6     /* 4320 */
    sys sys_comcat        0     /* 系统调用号为 4321 */
  .endm
```

最后加入自定义的系统调用 sys_comcat 于 sys_call_table 中， sys 为辅助宏，调用名后的数值 0 指示该系统调用所需的参数个数。

只能加在 sys_call_table 的最后，否则会扰乱标准的系统调用。

2. 实现系统调用

可以在 arch/mips/kernel/syscall.c 中给出一个实现，如：

```
asmlinkage void sys_comcat(void)
{
    printk(KERN_EMERG "This's comcat syscall\n");
}
```

3. 修改 include/asm-mips/unistd.h

根据你的内核位和支持的 ABI 版本修改相应的宏定义，32bit 内核需修改：

```
#define __NR_Linux_syscalls     320    ---------> 321        (The
number of linux syscalls)
#define __NR_O32_Linux_syscalls     320  ---------> 321  (The
number of linux O32 syscalls)
```

相应的在其上的宏集的最后加入：

```
.......
#define __NR_timerfd            (__NR_Linux + 318)
#define __NR_eventfd            (__NR_Linux + 319)
#define __NR_fallocate          (__NR_Linux + 320)
#define __NR_comcat             (__NR_Linux + 321)
```

4. 测试新的系统调用 sys_comcat

用重新编译后的内核启动，如下程序测试之：

```
.text
.globl  main
.ent    main

main:

    li   $2, 4321          /* sys_comcat 的系统的调用号 */
    syscall

.end    main
```

其中 MIPS 下系统调用的约定为：

```
v0:          用于置系统调用号
a0~a3:       置前四个参数，后面的参数用栈传
syscall      系统调用触发指令
```

编译 gcc cat.S -o cat

执行 ./cat

正确的话应有如下输出：

```
Message from syslogd@localhost at Wed Aug 29 13:15:37 2007 ...
localhost kernel: This's comcat syscall
```

Reference:

[1] See MIPS Run 2nd Edtion[M]. 北京：机械工业出版社，2007

[2] 张福新，陈怀临，MIPS 体系结构剖析、编程与实践.

[3] 龙芯 2E 用户手册 [S]