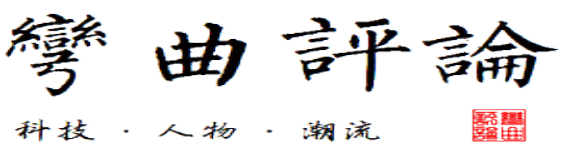

MIPS CPU 体系结构概述，Linux/MIPS内核

（下）

陈怀临，张福新

弯曲评论、中国科学院计算所

www.tektalk.cn www.ict.ac.cn

第二部分 Linux/MIPS核心剖析

1.硬件知识

* CPU 手册: http://www.mips.com
* 主板资料：相应的厂商.
* 背景知识:如PCI协议,中断概念等.

2.软件资源

* http://oss.sgi.com/linux , ftp://oss.sgi.com
* http://www.mips.com
* mailing lists:
  linux-mips@oss.sgi.com
  debian-mips@oss.sgi.com
* kernel cvs

sgi:

cvs -d :pserver:cvs@oss.sgi.com:/cvs login

(Only needed the first time you use anonymous CVS, the password is "cvs")

cvs -d :pserver:cvs@oss.sgi.com:/cvs co linux

另外sourceforge.net也有另一个内核树，似乎不如sgi的版本有影响.

* 经典书籍:
  * "Mips R4000 Microprocessor User's Manual",by Joe Heinrich
  * "See Mips Run",by Dominic Sweetman
  * "See Mips Run"(中文版）www.xtrj.org/smr.htm
* Jun Sun's mips porting guide: http://linux.junsun.net/porting-howto/
* 交叉编译指南: http://www.ltc.com/~brad/mips/mips-cross-toolchain.html
* Debian Mips port: http://www.debian.org/ports/mips
* 系统计算研究所网站: http://www.xtrj.org

3. mips kernel的一般介绍

(下面一些具体代码基于2.4.8的内核)

我们来跟随内核启动运行的过程看看mips内核有什么特别之处.

加电后,mips kernel从系统固件程序(类似bios,可能烧在eprom,flash中)得到控制之后(head.S),初始化内核栈,调用init_arch初始化硬件平台相关的代码.

init_arch(setup.c)首先监测使用的CPU(通过MIPS CPU的CP0控制寄存器PRID)确定使用的指令集和一些CPU参数,如TLB大小等.然后调用prom_init做一些底层参数初始化. prom_init是和具体的硬件相关的.

使用MIPS CPU的平台多如牛毛, 所以大家在arch/mips下面可以看到很多的子目录, 每个子目录是一个或者一系列相似的平台.这里的平台差不多可以理解成一块主板加上它的系统固件,其中很多还包括一些专用的显卡什么的硬件(比如一些工作站). 这些目录的主要任务是:

1. 提供底层板子上的一些重要信息,包括系统固件传递的参数,io的映射基地址,内存的大小的分布等.多数还包括提供早期的信息输入输出接口(通常是一个简单的串口驱动)以方便调试,因为pmon往往不提供键盘和显示卡的支持.?
2. 底层中断代码,包括中断控制器编程和中断的分派,应答等
3. pci子系统底层代码. 实现pci配置空间的读写,以及pci设备的中断,IO/Mem

空间的分配
4. 其它,特定的硬件.常见的有实时时钟等

这里关键是要理解这些硬件平台和熟悉的x86不同之处.笔者印象较深的有几个:

* item MIPS不象X86有很标准的硬件软件接口,而是五花八门,每个厂家有一套,因为它们很多是嵌入式系统或者专门的工作站.不象PC中,有了BIOS后用同一套的程序,就可以使用很多不同的主板和CPU.

MIPS中的'bios'常用的有pmon和yamon,都是开放源代码的软件。
很多开发板带的固件功能和PC BIOS很不一样,它们多数支持串口显示,或者网络下载和启动,以及类DEBUG的调试界面,但可能根本不支持显卡和硬盘,没有一般的基本'输入输出'功能.

* PCI系统和地址空间,总线等问题.

在x86中,IO空间用专门的指令访问,而PCI设备的内存空间和物理内存空间是相同的,也就是说,在CPU看来物理内存从地址0开始的话,在PCI设备看来也是一样的.反之,PCI设备的基地址寄存器设定的PCI存储地址,CPU用相同的物理地址访问就行了.

而在MIPS中就很不一样了,IO一般是memory map的,map到哪里就倚赖具体平台了.而PCI设备的地址空间和CPU所见的物理内存地址空间往往也不一样(bus address & physical address).所以mips kernel的iob/outb,以及bus_to_virt/virt_to_bus,phys_to_virt/virt_to_phys,ioremap等就要小心考虑.这些问题有时间笔者会对这些问题做专门的说明.

PCI配置空间的读写和地址空间映射的处理通常都是每个平台不一样的.
因为缺乏统一接口的BIOS,内核经常要自己做PCI设备的枚举,空间分配,中断分配.

* 中断系统.

PC中中断控制器先是有8259,后来是apic,而cpu的中断处理386之后好像也变化不大,相对统一.
mips CPU的中断处理方式倒是比较一致,但是主板上的控制器就乱七八糟了怎么鉴别中断源,怎么编程控制器等任务就得各自实现了.

总的说来,MIPS CPU的中断处理方式体现了RISC的特点:软件做事多,硬件尽量精简. 编程控制器,提供中断控制接口,dispatch中系?这一部分原来很混乱,大家各写各的,现在有人试图写一些比较统一的代码(实际上就是原来x86上用的controller/handler 抽象).

* 存储管理.
MIPS 是典型的RISC结构,它的存储管理单元做的事情比象x86这种机器少得多.例如,它的tlb是软件管理的,cache常常是需要系统程序干预的.而且,过多的CPU和主板变种使得这一部分非常复杂,容易出错.存储管理的代码主要在include/asm-mips和arch/mips/mm/目录下.
* 其它.
如时间处理,r4k以上的MIPS CPU提供count/compare寄存器,每隔几拍count增加,到和compare相等时发生时钟中断,这可以用来提供系统的时钟中断.但很多板子自己也提供其它的可编程时钟源.具体用什么就取决于开发者了.

init_arch后是loadmmu,初始化cache/tlb.代码在arch/mips/mm里.有人可能会问,在cache和tlb之前CPU怎么工作的?
在x86里有实模式,而MIPS没有,但它的地址空间是特殊的,分成几个不同的区域,每个区域中的地址在CPU里的待遇是不一样的,系统刚上电时CPU从地址bfc00000开始,那里的地址既不用tlb也不用cache,所以CPU能工作而不管cache和tlb是什么样子.当然,这样子效率是很低的,所以CPU很快就开始进行loadmmu. 因为MIPS CPU变种繁多,所以代码又臭又长. 主要不外是检测cache大小,选择相应的cache/tlb flush过程,还有一些memcpy/memset等的高效实现.这里还很容易出微妙的错误,软件管理tlb或者cache都不简单,要保证效率又要保证正确.在开发初期常常先关掉CPU的cache以便排除cache问题.

MMU初始化后,系统就直接跳转到init/main.c中的start_kernel,很快吧?
不过别高兴,start_kernel虚晃一枪,又回到arch/mips/kernel/setup.c,调用setup_arch,这回就是完成上面说的各平台相关的初始化了.

平台相关的初始化完成之后,mips内核和其它平台的内核区别就不大了,但也还有不少问题需要关注.如许多驱动程序可能因为倚赖x86的特殊属性(如IO端口,自动的cache一致性维护,显卡初始化等)而不能直接在MIPS下工作.

例如,能直接(用现有的内核驱动)在MIPS下工作的网卡不是很多,笔者知道的有intel

eepro100,AMD pcnet32 ,Tulip. 3com的网卡好像大多不能用.显卡则由于vga bios的问题,很少能直接使用.(常见的显卡都是为x86做的,它们常常带着一块rom,里面含有vga bios,PC的BIOS的初始化过程中发现它们的化就会先去执行它们以初始化显卡,然后才能很早地在屏幕上输出信息).而vga bios里面的代码一般是for x86,不能直接在mips CPU上运行.而且这些代码里常常有一些厂家相关的特定初始化,没有一个通用的代码可以替换.只有少数比较开放的厂家提供足够的资料使得内核开发人员能够跳过vga bios的执行直接初始化他的显卡,如matrox.

除此之外,也可能有其它的内核代码由于种种原因(不对齐访问,unsigned/signed等)不能使用,如一些文件系统(xfs?).

关于linux-mips内核的问题,在sgi的mailing list搜索或者提问比较有希望获得解决.如果你足够有钱,可以购买montivista的服务.http://www.mvista.com.

4.mips的异常处理

1.硬件

mips CPU的异常处理中,硬件做的事情很少,这也是RISC的特点. 和x86系统相比,有两点大不一样:

* 硬件不负责具体鉴别异常,CPU响应异常之后需要根据状态寄存器等来确定究竟发生哪个异常.有时候硬件会做一点简单分类,CPU能直接到某一类异常的处理入口.
* 硬件通常不负责保存上下文.例如系统调用,所有的寄存器内容都要由软件进行必要的保存.

各种主板的中断控制种类很多,需要根据中断控制器和连线情况来编程.

2.kernel实现

* 处理程序什么时候安装?

traps_init(arch/mips/kernel/traps.c,setup_arch之后start_kernel调用)

```
...
/* Copy the generic exception handler code to it's final destination. */
memcpy((void *)(KSEG0 + 0x80), &except_vec1_generic, 0x80);
memcpy((void *)(KSEG0 + 0x100), &except_vec2_generic, 0x80);
```

```
 memcpy((void *)(KSEG0 + 0x180), &except_vec3_generic, 0x80);
 flush_icache_range(KSEG0 + 0x80, KSEG0 + 0x200);
 /*
  * Setup default vectors
  */
 for (i = 0; i <= 31; i++)
 set_except_vector(i, handle_reserved);
 ...
```

* 装的什么?

except_vec3_generic(head.S) (除了TLB refill例外都用这个入口):

```
  /* General exception vector R4000 version. */
NESTED(except_vec3_r4000, 0, sp)
.set    noat
mfc0    k1, CP_CAUSE
andi    k1, k1, 0x7c  /* 从cause寄存器取出异常号 */
li      k0, 31<<2      beq     k1, k0, handle_vced /* 如果是vced,处理之*/
li     k0, 14><<2      beq     k1, k0, handle_vcei /* 如果是vcei,处理之*/
/* 这两个异常是和cache相关的,cache出了问题,不能再在这个cached的位置处理啦 */
la     k0, exception_handlers /* 取出异常处理程序表 */
addu    k0, k0, k1      lw      k0, (k0)  /*处理函数*/
nop     jr      k0       /*运行异常处理函数*/
nop
```

那个异常处理程序表是如何初始化的呢?

在traps_init中,大家会看到set_exception_vector(i,handler)这样的代码,

填的就是这张表啦.可是,如果你用souce insigh之类的东西去找那个handler,往往

就落空了,??怎么没有handle_ri,handle_tlbl..._?不着急,只不过是一个小trick,

还记得x86中断处理的handler代码吗? 它们是用宏生成的:

entry.S   ...

```
#define BUILD_HANDLER(exception,handler,clear,verbose)
.align  5;
NESTED(handle_##exception, PT_SIZE, sp);
.set    noat;
SAVE_ALL;  /* 保存现场,切换栈(如必要)*/
__BUILD_clear_##clear(exception); /*关中断?*/
```

```
.set    at;
__BUILD_##verbose(exception);
jal     do_##handler;    /*干活*/
move   a0, sp;
ret_from_exception;   /*回去*/
nop;
END(handle_##exception)       /*生成处理函数*/
BUILD_HANDLER(adel,ade,ade,silent)          /* #4  */
BUILD_HANDLER(ades,ade,ade,silent)          /* #5  */
BUILD_HANDLER(ibe,ibe,cli,verbose)          /* #6  */
BUILD_HANDLER(dbe,dbe,cli,silent)           /* #7  */
BUILD_HANDLER(bp,bp,sti,silent)             /* #9  */
```

认真追究下去,这里的一些宏是很重要的,象SAVE_ALL(include/asm/stackframe.h),
异常处理要高效,正确,这里要非常小心.这是因为硬件做的事情实在太少了.
别的暂时先不说了,下面我们来看外设中断(它是一种特殊的异常).
entry.S并没有用BUILD_HANDLER生成中断处理函数,因为它是平台相关的
就以笔者的板子为例,在arch/mips/algor/p6032/kernel/中(标准内核没有)
增加了p6032IRQ.S这个汇编文件,里面定义了一个p6032IRQ的函数,它负责
鉴别中断源,调用相应的中断控制器处理代码,而在同目录的irq.c->init_IRQ
中调用set_except_vector(0,p6032IRQ)填表(所有的中断都引发异常0,并在
cause寄存器中设置具体中断原因).

下面列出这两个文件以便解说:
p6032IRQ.s

algor p6032(笔者用的开发板)中断安排如下:

```
MIPS IRQ       Source
*     --------       ------
*           0       Software (ignored)
*           1       Software (ignored)
*           2       bonito interrupt (hw0)
*           3       i8259A interrupt (hw1)
*           4       Hardware (ignored)
*           5       Debug Switch
*           6       Hardware (ignored)
```

```
    *           7       R4k timer (what we use)

     .text
     .set    noreorder
     .set    noat
     .align  5
      NESTED(p6032IRQ, PT_SIZE, sp)

    SAVE_ALL /* 保存现场,切换堆栈(if usermode -> kernel mode)*/
    CLI     /* 关中断,mips有多种方法禁止响应中断,CLI用清status相应位
           的方法,如下:
            Move to kernel mode and disable interrupts.
            Set cp0 enable bit as sign that we're running
            on the kernel stack */
#define CLI
    mfc0    t0,CP0_STATUS;
    li      t1,ST0_CU0|0x1f;
    or      t0,t1;
    xori    t0,0x1f;
    mtc0    t0,CP0_STATUS

    .set    at

    mfc0    s0, CP0_CAUSE
    /* get irq mask,中断响应时cause寄存器指示
      哪类中断发生
      注意,MIPS CPU只区分8个中断源,并没有象
      PC中从总线读取中断向量号,所以每类中断
      的代码要自己设法定位中断
     */

    /* 挨个检查可能的中断原因 */
    /* First we check for r4k counter/timer IRQ. */
    andi    a0, s0, CAUSEF_IP7
    beq     a0, zero, 1f
```

```
andi   a0, s0, CAUSEF_IP3      # delay slot, check 8259 interrupt
/* Wheee, a timer interrupt. */
li      a0, 63
jal     do_IRQ
move    a1, sp
j       ret_from_irq
nop                     # delay slot

1:      beqz    a0,1f
andi   a0, s0, CAUSEF_IP2

/* Wheee, i8259A interrupt. */
/* p6032也使用8259来处理一些pc style的设备*/
jal     i8259A_irqdispatch      /* 调用8259控制器的中断分派代码*/
move   a0, sp               # delay slot

j       ret_from_irq
nop                     # delay slot

1:      beq     a0, zero, 1f
andi   a0, s0, CAUSEF_IP5

/* Wheee, bonito interrupt. */
/* bonito是6032板的北桥,它提供了一个中断控制器*/
jal     bonito_irqdispatch
move   a0, sp               # delay slot
j       ret_from_irq
nop                     # delay slot

1:      beqz    a0,1f
nop

/* Wheee, a debug interrupt. */
jal     p6032_debug_interrupt
move   a0, sp               # delay slot
```

```
    j       ret_from_irq
    nop                         # delay slot


    1:
    /* Here by mistake?  This is possible, what can happen
     * is that by the time we take the exception the IRQ
     * pin goes low, so just leave if this is the case.
     */
    j       ret_from_irq
    nop
END(p6032IRQ)
```

irq.c部分代码如下:

p6032中断共有四类:

begin{enumerate}

item timer中断,单独处理

item debug中断,单独处理

item 8259中断,由8259控制器代码处理

item bonito中断由bonito控制器代码处理

end{enumerate}

```
/* now mips kernel is using the same abstraction as x86 kernel,
   that is, all irq in the system are described in an struct
   array: irq_desc[]. Each item of a specific item records
   all the information about this irq,including status,action,
   and the controller that handle it etc. Below is the controller
   structure for bonito irqs,we can easily guess its functionality
   from its names.*/

hw_irq_controller bonito_irq_controller = {
    "bonito_irq",
    bonito_irq_startup,
```

```
    bonito_irq_shutdown,
    bonito_irq_enable,
    bonito_irq_disable,
    bonito_irq_ack,
    bonito_irq_end,
    NULL                    /* no affinity stuff for UP */
};

void
bonito_irq_init(u32 irq_base)
{
    extern irq_desc_t irq_desc[];
    u32 i;

    for (i= irq_base; i< P6032INT_END; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = NULL;
        irq_desc[i].depth = 1;
        irq_desc[i].handler = &bonito_irq_controller;
    }

    bonito_irq_base = irq_base;
}

/* 中断初始化,核心的数据结构就是irq_desc[]数组
   它的每个元素对应一个中断,记录该中断的控制器类型,处理函数,状态等
   关于这些可以参见对x86中断的分析*/

void __init init_IRQ(void)
{
    Bonito;

    /*
     * Mask out all interrupt by writing "1" to all bit position in
     * the interrupt reset reg.
```

```
     */
    BONITO_INTEDGE = BONITO_ICU_SYSTEMERR | BONITO_ICU_MASTERERR
        | BONITO_ICU_RETRYERR | BONITO_ICU_MBOXES;
    BONITO_INTPOL = (1 << (P6032INT_UART1-16))
        | (1 << (P6032INT_ISANMI-16))
        | (1 << (P6032INT_ISAIRQ-16))
        | (1 << (P6032INT_UART0-16));

    BONITO_INTSTEER = 0;
    BONITO_INTENCLR = ~0;

    /* init all controllers */
    init_generic_irq();
    init_i8259_irqs();
    bonito_irq_init(16);

    BONITO_INTSTEER |= 1 << (P6032INT_ISAIRQ-16);
    BONITO_INTENSET = 1 <<  (P6032INT_ISAIRQ-16);

    /* hook up the first-level interrupt handler */
    set_except_vector(0, p6032IRQ);

    ...
}

/*p6032IRQ发现一个bonito中断后调用这个*/
asmlinkage void
bonito_irqdispatch(struct pt_regs *regs)
{
    Bonito;

    int irq;
    unsigned long int_status;
    int i;
```

```
    /* Get pending sources, masked by current enables */
        /* 到底是哪个中断呢?从主板寄存器读*/
        int_status = BONITO_INTISR & BONITO_INTEN & ~(1 << (P6032INT_ISAIRQ-16))
                ;

        /* Scan all pending interrupt bits and execute appropriate actions */
        for (i=0; i<32 && int_status; i++) {
                if (int_status & 1<<i) {
                        irq = i + 16; /* 0-15 assigned to 8259int,16-48 bonito*/
                                /* Clear bit to optimise loop exit */
                                int_status &= ~(1<<i);
                        do_IRQ(irq,regs);

                }
        }

        return;
}
```

8259控制器的代码类似,不再列出.

更高层一点的通用irq代码在arch/mips/kernel/irq.c arch/mips/kernel/i8259.c

总之,p6032上一个中断的过程是:

1.外设发出中断,通过北桥在cpu中断引脚上(mips CPU有多个中断引脚)引起异常

2. cpu自动跳转到0x80000180的通用异常入口,根据cause寄存器查表找到中断处理函数入口p6032IRQ

3. p6032IRQ保存上下文,识别中断类别,把中断转交给相应的中断控制器

4. 中断控制器的代码进一步识别出具体的中断号,做出相应的应答并调用中断处理do_irq

现在还有不少平台没有使用这种irq_desc[],controller,action的代码,阅读的时候可能要注意.

下面把include/asm-mips/stackframe.h
对着注解一下,希望能说清楚一些.
(因为时间关系,笔者写的文档将主要以这种文件注解为主,加上笔者认为有用的背景知识或者分析.)

```
/*
 一些背景知识

 一.mips汇编有个约定(后来也有些变化,我们不管,o32,n32),32个通用寄存器不是一视同仁
  的,而是分成下列部分:
    寄存器号          符号名          用途
     0              始终为0     看起来象浪费,其实很有用
     1              at          保留给汇编器使用
     2-3             v0,v1       函数返回值
     4-7             a0-a3       前头几个函数参数
     8-15            t0-t7       临时寄存器,子过程可以不保存就使用
     24-25           t8,t9       同上
     16-23           s0-s7       寄存器变量,子过程要使用它必须先保存
                                 然后在退出前恢复以保留调用者需要的值
     26,27           k0,k1       保留给异常处理函数使用
     28              gp          global pointer;用于方便存取全局或者静态变量
     29              sp          stack pointer
     30              s8/fp       第9个寄存器变量;子过程可以用它做frame pointer
     31              ra          返回地址
   硬件上这些寄存器并没有区别(除了0号),区分的目的是为了不同的编译器产生的代码
   可以通用

  二. r4k MIPS CPU中和异常相关的控制寄存器(这些寄存器由协处理器cp0控制,有独立的
存取方法)有:
     1.status 状态寄存器
     31  28 27 26 25 24        16 15         8 7 6  5 4 3  2   1   0
    ------------------------------------------------------------------
    | cu0-3|RP|FR|RE| Diag Status|   IM7-IM0  |KX|SX|UX|KSU|ERL|EXL|IE|
```

其中KSU,ERL,EXL,IE位在这里很重要:

KSU: 模式位 00 -kernel 01--Supervisor 10--User

ERL: error level,0->normal,1->error

EXL: exception level,0->normal,1->exception,异常发生是EXL自动置1

IE: interrupt Enable, 0 -> disable interrupt,1->enable interrupt

(IM位则可以用于enbale/disable具体某个中断,ERL||EXL=1 也使得中断不能响应)

系统所处的模式由KSU,ERL,EXL决定:

User mode: KSU = 10 && EXL=0 && ERL=0

Supervisor mode(never used): KSU=01 && EXL=0 && ERL=0

Kernel mode: KSU=00 || EXL=1 || ERL=1

2.cause寄存器

31 30 29 28 27 16 15 8 7 6 2 1 0

|BD|0 | CE | 0 | IP7 - IP0 |0|Exc code | 0 |

异常发生时cause被自动设置

其中:

BD指示最近发生的异常指令是否在delay slot中

CE发生coprocessor unusable异常时的coprocessor编号(mips有4个cp)

IP: interrupt pending, 1->pending,0->no interrupt,CPU有6个中断

引脚,加上两个软件中断(最高两个)

Exc code:异常类型,所有的外设中断为0,系统调用为8,...

3.EPC

对一般的异常,EPC包含:

. 导致异常的指令地址(virtual)

or. if 异常在delay slot指令发生,该指令前面那个跳转指令的地址

当EXL=1时,处理器不写EPC

4.和存储相关的:

context,BadVaddr,Xcontext,ECC,CacheErr,ErrorEPC

以后再说

一般异常处理程序都是先保存一些寄存器,然后清除EXL以便嵌套异常,

清除KSU保持核心态,IE位看情况而定;处理完后恢复一些保存内容以及CPU状态

```
  */

/* SAVE_ALL 保存所有的寄存器,分成几个部分,方便不同的需求选用*/
/*保存AT寄存器,sp是栈顶PT_R1是at寄存器在pt_regs结构的偏移量
  .set xxx是汇编指示,告诉汇编器要干什么,不要干什么,或改变状态
 */
#define SAVE_AT                                 \
        .set    push;                   \
        .set    noat;                   \
        sw      $1, PT_R1(sp);                  \
        .set    pop

/*保存临时寄存器,以及hi,lo寄存器(用于乘法部件保存64位结果)
 可以看到mfhi(取hi寄存器的值)后并没有立即保存,这是因为
 流水线中,mfhi的结果一般一拍不能出来,如果下一条指令就想
 用v1则会导致硬件停一拍,这种情况下让无关的指令先做可以提高
 效率.下面还有许多类似的例子
 */
#define SAVE_TEMP                               \
        mfhi    v1;                     \
        sw      $8, PT_R8(sp);                  \
        sw      $9, PT_R9(sp);                  \
        sw      v1, PT_HI(sp);                  \
        mflo    v1;                     \
        sw      $10,PT_R10(sp);                 \
        sw      $11, PT_R11(sp);                \
        sw      v1,  PT_LO(sp);                 \
        sw      $12, PT_R12(sp);                \
        sw      $13, PT_R13(sp);                \
        sw      $14, PT_R14(sp);                \
        sw      $15, PT_R15(sp);                \
        sw      $24, PT_R24(sp)

/* s0-s8 */
#define SAVE_STATIC                             \
```

```
        sw      $16, PT_R16(sp);            \
        sw      $17, PT_R17(sp);            \
        sw      $18, PT_R18(sp);            \
        sw      $19, PT_R19(sp);            \
        sw      $20, PT_R20(sp);            \
        sw      $21, PT_R21(sp);            \
        sw      $22, PT_R22(sp);            \
        sw      $23, PT_R23(sp);            \
        sw      $30, PT_R30(sp)

#define __str2(x) #x
#define __str(x) __str2(x)



/*ok,下面对这个宏有冗长的注解*/
#define save_static_function(symbol)                                \
__asm__ (                                                   \
        ".globl\t" #symbol "\n\t"                               \
        ".align\t2\n\t"                                     \
        ".type\t" #symbol ", @function\n\t"                         \
        ".ent\t" #symbol ", 0\n"                                \
        #symbol":\n\t"                                      \
        ".frame\t$29, 0, $31\n\t"                               \
        "sw\t$16,"__str(PT_R16)"($29)\t\t\t# save_static_function\n\t"  \
        "sw\t$17,"__str(PT_R17)"($29)\n\t"                          \
        "sw\t$18,"__str(PT_R18)"($29)\n\t"                          \
        "sw\t$19,"__str(PT_R19)"($29)\n\t"                          \
        "sw\t$20,"__str(PT_R20)"($29)\n\t"                          \
        "sw\t$21,"__str(PT_R21)"($29)\n\t"                          \
        "sw\t$22,"__str(PT_R22)"($29)\n\t"                          \
        "sw\t$23,"__str(PT_R23)"($29)\n\t"                          \
        "sw\t$30,"__str(PT_R30)"($29)\n\t"                          \
        ".end\t" #symbol "\n\t"                                 \
        ".size\t" #symbol",. - " #symbol)
```

```
/* Used in declaration of save_static functions.  */
#define static_unused static __attribute__((unused))

/*以下这一段涉及比较微妙的问题,没有兴趣可以跳过*/

/* save_static_function宏是一个令人迷惑的东西,它定义了一个汇编函数,保存s0-s8
   可是这个函数没有返回!实际上,它只是一个函数的一部分:
   在arch/mips/kernel/signal.c中有:
      save_static_function(sys_rt_sigsuspend);
      static_unused int
      _sys_rt_sigsuspend(struct pt_regs regs)
      {
            sigset_t *unewset, saveset, newset;
                  size_t sigsetsize;
   这里用save_static_function定义了sys_rt_sigsuspend,而实际上如果
   你调用sys_rt_sigsuspend的话,它保存完s0-s8后,接着就调用_sys_rt_sigsuspend!
   看它链接后的反汇编片段:
     80108cc8 <sys_rt_sigsuspend>:
     80108cc8:       afb00058        sw      $s0,88($sp)
     80108ccc:       afb1005c        sw      $s1,92($sp)
     80108cd0:       afb20060        sw      $s2,96($sp)
     80108cd4:       afb30064        sw      $s3,100($sp)
     80108cd8:       afb40068        sw      $s4,104($sp)
     80108cdc:       afb5006c        sw      $s5,108($sp)
     80108ce0:       afb60070        sw      $s6,112($sp)
     80108ce4:       afb70074        sw      $s7,116($sp)
     80108ce8:       afbe0090        sw      $s8,144($sp)

     80108cec <_sys_rt_sigsuspend>:
     80108cec:       27bdffc8        addiu   $sp,$sp,-56
     80108cf0:       8fa80064        lw      $t0,100($sp)
     80108cf4:       24030010        li      $v1,16
     80108cf8:       afbf0034        sw      $ra,52($sp)
     80108cfc:       afb00030        sw      $s0,48($sp) ---> notice
```

```
80108d00:    afa40038    sw    $a0,56($sp)
80108d04:    afa5003c    sw    $a1,60($sp)
80108d08:    afa60040    sw    $a2,64($sp)
...
```

用到save_static_function的地方共有4处:

```
signal.c:save_static_function(sys_sigsuspend);
signal.c:save_static_function(sys_rt_sigsuspend);
syscall.c:save_static_function(sys_fork);
syscall.c:save_static_function(sys_clone);
```

我们知道s0-s8如果在子过程用到,编译器本来就会保存/恢复它的(如上面的s0),
那为何要搞这个花招呢?笔者分析之后得出如下结论:
(警告:以下某些内容是笔者的推测,可能不完全正确)

先看看syscall的处理,syscall也是mips的一种异常,异常号为8.上次我们说
了一般异常是如何工作的,但在handle_sys并非用BUILD_HANDLER生成,而是在
scall_o23.S中定义,因为它又有其特殊之处.

1.缺省情况它只用了SAVE_SOME,并没有保存at,t*,s*等寄存器,因为syscall
是由应用程序调用的,不象中断,任何时候都可以发生,所以一般编译器就可以
保证不会丢数据了(at,t*的值应该已经无效,s*的值会被函数保存恢复).
这样可以提高系统调用的效率

2.它还得和用户空间打交道(取参数,送数据)

还有个别系统调用需要在特定的时候手工保存s*寄存器,如上面的几个.为什么呢?
对sigsuspend来说,它将使进程在内核中睡眠等待信号到来,信号来了之后将直接
先回到进程的信号处理代码,而信号处理代码可能希望看到当前进程的寄存器
(sigcontext),这是通过内核栈中的pt_regs结构获得的,所以内核必需把s*寄存器
保存到pt_regs中.对于fork的情况,则似乎是为了满足vfork的要求.(vfork时,子进程
不拷贝页表(即和父进程完全共享内存),注意,连copy-on-write都没有!父进程挂起
一直到子进程不再使用它的资源(exec或者exit)).fork 系统调用使用ret_from_fork
返回,其中调用到了RESTORE_ALL_AND_RET(entry.S),需要恢复s*.

这里还有一个很容易混乱的地方: 在scall_o32.S和entry.S中有几个函数(汇编)是同名
的,如restore_all,sig_return等.总体来说scall_o32.S中是对满足o32(old 32bit)汇编
约定的系统调用处理,可以避免保存s*,而entry.S中是通用的,保存/恢复所由寄存器
scall_o32.S中也有一些情况需要保存静态寄存器s*,此时它就会到ret_from_syscall

而不是本文件中的o32_ret_from_syscall返回了,两者的差别就是恢复的寄存器数目不同.scall_o32.S中一些错误处理直接用ret_from_syscall返回,笔者怀疑会导致s*寄存器被破坏,有机会请各路高手指教.

好了,说了一通系统调用,无非是想让大家明白内核中寄存器的保存恢复过程,以及为了少做些无用功所做的努力.下面看为什么要save_static_function:为了避免s0寄存器的破坏.

如果我们使用

```
sys_rt_sigsuspend()
{  ..
   save_static;
   ...
}
```

会有什么问题呢,请看,

Nasty degree - 3 days of tracking.

The symptom was pthread cannot be created. In the end the caller will get a BUS error.

What exactly happened has to do with how registers are saved. Below attached is the beginning part of sys_sigsuspend() function. It is easy to see that s0 is saved into stack frame AFTER its modified. Next time when process returns to userland, the s0 reg will be wrong!

So the bug is either

1) that we need to save s0 register in SAVE_SOME and not save it in save_static; or that

2) we fix compiler so that it does not use s0 register in that case (it does the same thing for sys_rt_sigsuspend)

I am sure Ralf will have something to say about it. :-) In any case, I attached a patch for 1) fix.

```
sys_sigsuspend(struct pt_regs regs)
{
      8008e280:   27bdffc0       addiu   $sp,$sp,-64
      8008e284:   afb00030       sw      $s0,48($sp)
            sigset_t *uset, saveset, newset;

             save_static(&regs);
      8008e288:   27b00040       addiu   $s0,$sp,64   /* save_static时
                                        s0已经破坏*/
      8008e28c:   afbf003c       sw      $ra,60($sp)
      8008e290:   afb20038       sw      $s2,56($sp)
      8008e294:   afb10034       sw      $s1,52($sp)
      8008e298:   afa40040       sw      $a0,64($sp)
      8008e29c:   afa50044       sw      $a1,68($sp)
      8008e2a0:   afa60048       sw      $a2,72($sp)
      8008e2a4:   afa7004c       sw      $a3,76($sp)
      8008e2a8:   ae100058       sw      $s0,88($s0)
      8008e2ac:   ae11005c       sw      $s1,92($s0)

#ifdef CONFIG_SMP
#  define GET_SAVED_SP                              \
            mfc0    k0, CP0_CONTEXT;                \
            lui     k1, %hi(kernelsp);              \
            srl     k0, k0, 23;                     \
           sll     k0, k0, 2;                      \
            addu    k1, k0;                         \
            lw      k1, %lo(kernelsp)(k1);

#else
#  define GET_SAVED_SP                              \
/*实际上就是k1 = kernelsp, kernelsp保存当前进程的内核栈指针 */
           lui     k1, %hi(kernelsp);              \
           lw      k1, %lo(kernelsp)(k1);
#endif
```

```
/*判断当前运行态,设置栈顶sp
  保存寄存器--参数a0-a3:4-7,返回值v0-v1:2-3,25,28,31以及一些控制寄存器,
  */
#define SAVE_SOME                                       \
            .set    push;                       \
            .set    reorder;                     \
            mfc0    k0, CP0_STATUS;              \
            sll     k0, 3;     /* extract cu0 bit */ \
            .set    noreorder;                   \
            bltz    k0, 8f;                     \
             move   k1, sp;                      \
            .set    reorder;                     \
            /* Called from user mode, new stack. */  \
             GET_SAVED_SP                        \
8:                                              \
            move    k0, sp;                     \
            subu    sp, k1, PT_SIZE;            \
            sw      k0, PT_R29(sp);             \
             sw     $3, PT_R3(sp);              \
            sw      $0, PT_R0(sp);              \
            mfc0    v1, CP0_STATUS;             \
            sw      $2, PT_R2(sp);              \
            sw      v1, PT_STATUS(sp);          \
            sw      $4, PT_R4(sp);              \
            mfc0    v1, CP0_CAUSE;              \
            sw      $5, PT_R5(sp);              \
            sw      v1, PT_CAUSE(sp);           \
            sw      $6, PT_R6(sp);              \
            mfc0    v1, CP0_EPC;                \
            sw      $7, PT_R7(sp);              \
            sw      v1, PT_EPC(sp);             \
            sw      $25, PT_R25(sp);            \
            sw      $28, PT_R28(sp);            \
            sw      $31, PT_R31(sp);            \
```

```
        ori     $28, sp, 0x1fff;            \
        xori    $28, 0x1fff;                \
        .set    pop

#define SAVE_ALL                            \
        SAVE_SOME;                          \
        SAVE_AT;                            \
        SAVE_TEMP;                          \
        SAVE_STATIC

#define RESTORE_AT                          \
        .set    push;                       \
        .set    noat;                       \
        lw      $1,  PT_R1(sp);             \
        .set    pop;

#define RESTORE_TEMP                        \
        lw      $24, PT_LO(sp);             \
        lw      $8, PT_R8(sp);              \
        lw      $9, PT_R9(sp);              \
        mtlo    $24;                        \
        lw      $24, PT_HI(sp);             \
        lw      $10,PT_R10(sp);             \
        lw      $11, PT_R11(sp);            \
        mthi    $24;                        \
        lw      $12, PT_R12(sp);            \
        lw      $13, PT_R13(sp);            \
        lw      $14, PT_R14(sp);            \
        lw      $15, PT_R15(sp);            \
        lw      $24, PT_R24(sp)

#define RESTORE_STATIC                      \
        lw      $16, PT_R16(sp);            \
        lw      $17, PT_R17(sp);            \
        lw      $18, PT_R18(sp);            \
        lw      $19, PT_R19(sp);            \
```

```
        lw      $20, PT_R20(sp);                \
        lw      $21, PT_R21(sp);                \
        lw      $22, PT_R22(sp);                \
        lw      $23, PT_R23(sp);                \
        lw      $30, PT_R30(sp)

#if defined(CONFIG_CPU_R3000) || defined(CONFIG_CPU_TX39XX)

#define RESTORE_SOME                                    \
        .set    push;                           \
        .set    reorder;                        \
        mfc0    t0, CP0_STATUS;                 \
        .set    pop;                            \
        ori     t0, 0x1f;                       \
        xori    t0, 0x1f;                       \
        mtc0    t0, CP0_STATUS;                 \
        li      v1, 0xff00;                     \
        and     t0, v1;                         \
        lw      v0, PT_STATUS(sp);              \
        nor     v1, $0, v1;                     \
        and     v0, v1;                         \
        or      v0, t0;                         \
        mtc0    v0, CP0_STATUS;                 \
        lw      $31, PT_R31(sp);                \
        lw      $28, PT_R28(sp);                \
        lw      $25, PT_R25(sp);                \
        lw      $7,  PT_R7(sp);                 \
        lw      $6,  PT_R6(sp);                 \
        lw      $5,  PT_R5(sp);                 \
        lw      $4,  PT_R4(sp);                 \
        lw      $3,  PT_R3(sp);                 \
        lw      $2,  PT_R2(sp)

#define RESTORE_SP_AND_RET                              \
        .set    push;                           \
        .set    noreorder;                      \
```

```
        lw      k0, PT_EPC(sp);                 \
        lw      sp,  PT_R29(sp);                \
        jr      k0;                             \
         rfe;                                   \
         ^^^^^
/* 异常返回时,把控制转移到用户代码和把模式从内核态改为用户态要同时完成
   如果前者先完成,用户态指令有机会以内核态运行导致安全漏洞;
   反之则会由于用户态下不能修改状态而导致异常
   r3000以前使用rfe(restore from exception)指令,这个指令把status寄存器
   状态位修改回异常发生前的状态(利用硬件的一个小堆栈),但不做跳转.我们使用一个
   技巧来完成要求:在一个跳转指令的delay slot中放rte.因为delay slot的指令
   是一定会做的,跳转完成时,status也恢复了.
   MIPS III(r4000)以上的指令集则增加了eret指令来完成整个工作: 它清除
   status寄存器的EXL位并跳转到epc指定的位置.
*/

        .set    pop

#else

#define RESTORE_SOME                            \
        .set    push;                           \
        .set    reorder;                        \
        mfc0    t0, CP0_STATUS;                 \
        .set    pop;                            \
        ori     t0, 0x1f;                       \
        xori    t0, 0x1f;                       \
        mtc0    t0, CP0_STATUS;                 \
        li      v1, 0xff00;                     \
        and     t0, v1;                         \
        lw      v0, PT_STATUS(sp);              \
        nor     v1, $0, v1;                     \
        and     v0, v1;                         \
        or      v0, t0;                         \
        mtc0    v0, CP0_STATUS;                 \
```

```
        lw      v1, PT_EPC(sp);             \
        mtc0    v1, CP0_EPC;                \
        lw      $31, PT_R31(sp);            \
        lw      $28, PT_R28(sp);            \
        lw      $25, PT_R25(sp);            \
        lw      $7,  PT_R7(sp);             \
        lw      $6,  PT_R6(sp);             \
        lw      $5,  PT_R5(sp);             \
        lw      $4,  PT_R4(sp);             \
        lw      $3,  PT_R3(sp);             \
        lw      $2,  PT_R2(sp)

#define RESTORE_SP_AND_RET                  \
        lw      sp,  PT_R29(sp);            \
        .set    mips3;                      \
        eret;                               \
        .set    mips0

#endif

#define RESTORE_SP                          \
        lw      sp,  PT_R29(sp);            \

#define RESTORE_ALL                         \
        RESTORE_SOME;                       \
        RESTORE_AT;                         \
        RESTORE_TEMP;                       \
        RESTORE_STATIC;                     \
        RESTORE_SP

#define RESTORE_ALL_AND_RET                 \
        RESTORE_SOME;                       \
        RESTORE_AT;                         \
        RESTORE_TEMP;                       \
        RESTORE_STATIC;                     \
        RESTORE_SP_AND_RET
```

```
/*
 * Move to kernel mode and disable interrupts.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define CLI                                     \
        mfc0    t0,CP0_STATUS;                  \
        li      t1,ST0_CU0|0x1f;                \
        or      t0,t1;                          \
        xori    t0,0x1f;                        \
        mtc0    t0,CP0_STATUS

/*
 * Move to kernel mode and enable interrupts.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define STI                                     \
        mfc0    t0,CP0_STATUS;                  \
        li      t1,ST0_CU0|0x1f;                \
        or      t0,t1;                          \
        xori    t0,0x1e;                        \
        mtc0    t0,CP0_STATUS

/*
 * Just move to kernel mode and leave interrupts as they are.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define KMODE                                   \
        mfc0    t0,CP0_STATUS;                  \
        li      t1,ST0_CU0|0x1e;                \
        or      t0,t1;                          \
        xori    t0,0x1e;                        \
        mtc0    t0,CP0_STATUS

#endif /* __ASM_STACKFRAME_H */
```

下面是笔者在为godson CPU的页面可执行保护功能增加内核支持时分析linux-mips mmu实现的一些笔记。也许第5节对整个工作过程的分析会有些用,其它语焉不详的东西多数只是对笔者本人有点用.

首先的,关键的,要明白MIPS CPU的tlb是软件管理的,cache也不是透明的,具体的参见它们的用户手册.

(for sgi-cvs kernel 2.4.17)

1. mmu context

cpu用8位asid来区分tlb表项所属的进程,但是进程超过256个怎么办?

linux实现的思想是软件扩展,每256个一组,TLB任何时候只存放同一组的asid因此不会冲突. 从一组的某个进程切换到另一组时,把tlb刷新

ASID switch

include/asm/mmu_context.h:

asid_cache:

8bit physical asid + software extension, the software extension bits are used as a version; this records the newest asid allocated,while process->mm->context records its own version.

get_new_mmu_context:

asid_cache++, if increasement lead to change of software extension part then flush icache & tlb to avoid conflicting with old versions.

asid_cache = 0 reserved to represent no valid mmu context case,so the first asid_cache version start from 0x100.

switch_mm:

if asid version of new process differs from current process',get a new context for it.(it's safe even if it gets same 8bit asid as previous because this process' tlb entries must have been flushed at the time of version increasement)

set entryhi,install pgd

activate_mm:

get new asid,set to entryhi,install pgd.

2. pte bits

页表的内容和TLB表项关系

entrylo[01]:

3130 29 6 5 3 2 1 0

```
    ------------------------------------------
    |   |     PFN                | C  |D|V|G|
    ------------------------------------------
```

r4k pte:

```
    31               12 111098 7 6 5  3 2 1 0
    ------------------------------------------
    |    PFN           | C  |D|V|G|B|M|A|W|R|P|
    ------------------------------------------
```

C: cache attr.
D: Dirty
V: valid
G: global
B: R4K_BUG
M: Modified
A: Accessed
W: Write
R: Read
P: Present
(last six bits implemented in software)

godson entrylo:

bit 30 is used as execution protect bit E,only bit25-6 are used as PFN.
instruction fetch from a page has E cleared lead to address error exception.

godson pte:

```
    31               12 111098 7 6 5  3 2 1 0
    ------------------------------------------
    |    PFN           | C  |D|V|G|E|M|A|W|R|P|
    ------------------------------------------
```

E: software implementation of execute protection.Page is executable when E is set,non-executable otherwise.(Notice,it is different from hardware bit 30 in entrylo)

3. actions dealing with pte

pte_page: get page struct from its pte value

pte_{none,present,read,write,dirty,young}: get pte status,use software bits
pte_wrprotect: &= ~(_PAGE_WRITE | _PAGE_SILENT_WRITE)
pte_rdprotect: &= ~(_PAGE_READ | _PAGE_SILENT_READ)
pte_mkclean: &= ~(_PAGE_MODIFIED | _PAGE_SILENT_WRITE)
pte_mkold: &= ~(_PAGE_ACCESSED | _PAGE_SILENT_READ)
pte_mkwrite: |= _PAGE_WRITE && if (_PAGE_MODIFIED) |= _PAGE_SILENT_WRITE
pte_mkread: |= _PAGE_READ && if (_PAGE_ACCESSED) |= _PAGE_SILENT_READ
pte_mkdirty: |= _PAGE_MODIFIED && if (_PAGE_WRITE) |= _PAGE_SILENT_WRITE
pte_mkyoung: |= _PAGE_ACCESSED && if (_PAGE_READ) |= _PAGE_SILENT_READ
pgprot_noncached: (&~CACHE_MASK) | (_CACHE_UNCACHED)
mk_pte(page,prot): (unsigned long) (page - memmap) << PAGE_SHIFT | prot
mk_pte_phys(physpage,prot): physpage | prot
pte_modify(pte,prot): (pte & _PAGE_CHG_MASK) | newprot
page_pte_{prot}: unused?
set_pte: *ptep = pteval
pte_clear: set_pte(ptep,__pte(0));

ptep_get_and_clear

pte_alloc/free

4. exceptions
tlb refill exception(0x80000000):
(1) get badvaddr,pgd
(2) pte table ptr = badvaddr>>22 < 2 + pgd ,
(3) get context,offset = context >> 1 & 0xff8 (bit 21-13 + three zero),
(4) load offset(pte table ptr) and offset+4(pte table ptr),
*(5) right shift 6 bits,write to entrylo[01],
(6) tlbwr
tlb modified exception(handle_mod):
(1) load pte,
*(2) if _PAGE_WRITE set,set ACCESSED | MODIFIED | VALID | DIRTY,
reload tlb,tlbwi
else DO_FAULT(1)
tlb load exception(handle_tlbl):

(1) load pte
(2) if _PAGE_PRESENT && _PAGE_READ, set ACCESSED | VALID
else DO_FAULT(0)
tlb store exception(handle_tlbs):
(1) load pte
*(2) if _PAGE_PRESENT && _PAGE_WRITE,set ACCESSED | MODIFIED | VALID | DIRTY
else DO_FAULT(1)

items marked with * need modification.

5. protection_map
all _PXXX map to page_copy? Although vm_flags will at last make pte writeable as needed,but will this be inefficient? it seems that alpha is not doing so.

mm setup/tear down:
on fork,copy_mm:
allocate_mm,
memcpy(new,old)
slow path
mm_init-->pgd_alloc-->pgd_init-->all point to invalid_pte
-->copy kseg pgds from init_mm
fast path: what's the content of pgd?
--> point to invalid_pte too,see clear_page_tables
dup_mmap->copy_page_range-->alloc page table entries and do cow if needed.
copy_segmens--null
init_new_context--set mm->context=0(allocate an array for SMP first)
on exec(elf file),load_elf_binary:
flush_old_exec:
exec_mmap
exit_mmap(old_mm)
free vm_area_struct
zap_page_range: free pages
clear_page_tables
pgd_clear: do nothing
pmd_clear: set to invalid_pte

pte_clear: set to zero
mm_alloc
initialize new mm(init_new_contex,add to list,activate it)
mmput(oldmm)
setup_arg_pages:
initialize stack segment. mm_area_struct for stack segment is setup here.
load elf image into the correct location in memory
elf_prot generated from eppnt->p_flags
elf_map(..,elf_prot,..)
do_mmap

a typical session for a user page to be read then written:
(1) user allocates the space
(2) kernel call do_mmap/do_brk, vm_area_struct created
(3) user tries to read
(4) tlb refill exception occurs,invalid_pte_table's entry is loaded into tlb
(5) tlbl exception occurs,
do_page_fault(0)->handle_mm_fault(allocate pte_table)->handle_pte_fault
-->do_no_page-->map to ZERO page,readonly,set_pte,update_mmu_cache
(update_mmu_cache put new pte to tlb,NEED change for godson)
(6) read done,user tries to write
(7) tlbs exception occurs(suppose the tlb entry is not yet kicked out)
because pte is write protected,do_page_fault(1) called.
handle_mm_fault(find out the pte)-->handle_pte_fault->do_wp_page
-->allocate page,copy page,break_cow-->make a writeable pte,
-->establish_pte-->write pte and update_mmu_cache
(8) write done.

above has shown that handle_mm_fault doesn't care much about what the page_prot is. (Of course,it has to be reasonable)
What really matters is vm_flags,it will decide whether an access is valid

6. do_page_fault
seems ok

7. swapping
  seems ok

8. adding execution protection
 2002.3.16:
    TLB execute protection bit support.
     1. generic support
     idea:
      use bit 5 in pte to maintain a software version of _PAGE_EXEC
      modify TLB refill code to reflect it into hardware bit(bit 30)
     affected files:
      include/asm/pgtable.h:
       define _PAGE_EXEC
       change related PAGE_XXX macros and protection_map
       add pte_mkexec/pte_exprotect
       add godson_mkexec/godson_mkprotect
      arch/mips/mm/tlbex-r4k.S:
       tlb_refill exception & PTE_RELOAD macro:
       test bit 5 and translated it into bit30 in entrylo
       using godson's cp0 register 23/24 as temporary store place
       Note: bit5 and bit30 have adverse meaning,bit5 set==bit30
            cleared==page executable,
      arch/mips/mm/tlb-r4k.c:
       update_mmu_cache:
          test bit 5 and translated it into bit30 in entrylo
       implement godson_mkexec/godson_exprotect
      arch/mips/config.in:
       add option CONFIG_CPU_HAS_EXECUTE_PROTECTION
     2. non-executable stack support
      interface:
       by default no protection is taken,To take advantage of
       this support,one should call sysmips syscall to set the
       flag bit and then execute the target program.
      affected files:
       include/asm/processor.h:
          define MF_STACK_PROTECTION flag

fs/exec.c:

judge which protection to use

arch/mips/kernel/signal.c:

enable/disable execute for signal trampoline

arch/mips/math-emu/cp1emu.c:

enable/disable execute for delay slot emulation trampoline

arch/mips/kernel/sysmips.c:

handle MF_STACK_PROTECTION