MIPS 存储管理结构及Linux代码分析

CurrentVersion: 0.1
Date: 2008-03-23
Author: Jack Tan <jack.tan@windriver.com>

版本历史

版本状态	作 者	参与者	起止日期	备注
0.1	Jack Tan		08-03-23	初始化
0.2	Jack Tan		08-08-17	完成框架
0.3	Jack Tan		08-09-24	完成草稿

1. MIPS 存储管理概述

1.1 虚拟地址空间

与 x86, ARM, PowerPC 以及 SPARC 不同，MIPS 在体系结构的规范里，对虚拟地址空间进行了划分。MIPS32 和 MIPS64 分别对 32 位和 64 位的情形进行了规定，要留意的是 MIPS64 的划分兼容于 MIPS32，即当 MIPS64 的处理器运行于 32 位模式时，其虚拟地址空间的划分“看上去”是和 32 位的一样的。

1.1.1 32 位虚拟地址空间

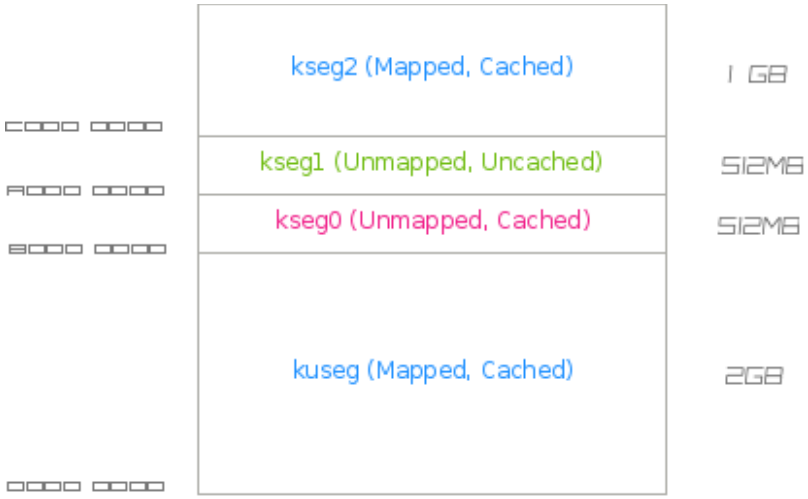

Figure 1.1: 32 bit Virtual Address Space

其中，kuseg 给用户态程序使用，访问这个区域，虚拟地址会经 TLB 转换 (Mapped) 同时被访问之数据可被缓存于 cache (Cached)

kseg0, kseg1 给操作系统使用，访问这两个区域，虚拟地址都不会经 TLB 转换，其皆固定映射到物理地址 0x0000 0000 ~ 0x1FFF FFFF （低 512MB）处。他们的差别在于：

访问 kseg1 的数据不会缓存到 cache 中 (Uncached)，这个区域往往用于 I/O （如对设备寄存器的读写）。

kseg2 亦是给操作系统使用，这个区域既经 TLB 映射、数据亦能被缓存于 cache

用户态程序不能访问 kseg0, kseg1, kseg2，否则会引发 Address Error 异常。

因此，很自然的，用户空间和内核空间的划分不会是由 OS 设计为 3G + 1G 。OS 的设计需要符合 MIPS 规范的约定，则用户空间为 2G ，而内核空间可以有 2G 。

1.1.2 64 位虚拟地址空间

MIPS64 时，虚拟地址空间划分有些复杂。MIPS64 规范将 64 位地址空间划分为 4 段，分别为 xuseg, xsseg, xkphys, xkseg。其中为了保持与 MIPS32 的兼容，xkseg 最顶端的 2G 对应于 MIPS32 的 kseg2, kseg1 和 kseg0，xuseg 的低 2G 则对应于 MIPS32 2G 大小的 kuseg。
需要注意的是 MIPS 的具体实现，可以选择实现的虚拟地址的位数，这个值可以通过如下步骤获取之：

1. 写所有位为 1 到 CP0 之 EntryHi
2. 读取 EntryHi 的值

EntryHi 之 VPN2 中，值为 1 的位则说明该位是被实现的

一般 MIPS64 的处理器，往往实现 40 位的虚拟地址。

64 位 Linux 设计时，内核运行在 xkseg 之 kseg0，对物理内存的直接访问则不像 32 位那样使用 kseg1 了，其使用 xkphys。xkphys 的虚址都是直接映射到物理地址的，访问这个区域是不会走 TLB 的，至于对访问的数据 cache 与否，则通过位 VA[61:59] 来控制，例如某个 MIPS64 的实现，使用 36 位的物理地址，则：
访问 0x9000 0000 0000 0000 ~ 0x9000 000F FFFF FFFF 时，其数据是不会被缓存的，因为该段虚拟地址的 61 ~ 59 的值为 2，对应于 Cache 一致性属性的 Uncached。更详细的，MIPS64 规范对 xkphys 作了详细的区分：

xkphys 大小为 0x4000 0000 0000 0000，其以 0x0800 0000 0000 0000 为单位划分为 8 个小段，分别为：

0xB800 0000 0000 0000 ~ 0xBFFF FFFF FFFF FFFF
0xB000 0000 0000 0000 ~ 0xB7FF FFFF FFFF FFFF
0xA800 0000 0000 0000 ~ 0xAFFF FFFF FFFF FFFF
0xA000 0000 0000 0000 ~ 0xA7FF FFFF FFFF FFFF
0x9800 0000 0000 0000 ~ 0x9FFF FFFF FFFF FFFF

0x9000 0000 0000 0000 ~ 0x97FF FFFF FFFF FFFF
0x8800 0000 0000 0000 ~ 0x88FF FFFF FFFF FFFF
0x8000 0000 0000 0000 ~ 0x87FF FFFF FFFF FFFF

VA[61:59] 分别为 7, 6, 5, 4, 3, 2, 1, 0，对应 Cache Coherency Attribute 的 8 个值。

对于实现 36 位物理地址的处理器，访问 xkphys 时，CPU 直接将 VA[63:36] 置为 0 即为该虚拟地址对应的物理地址。

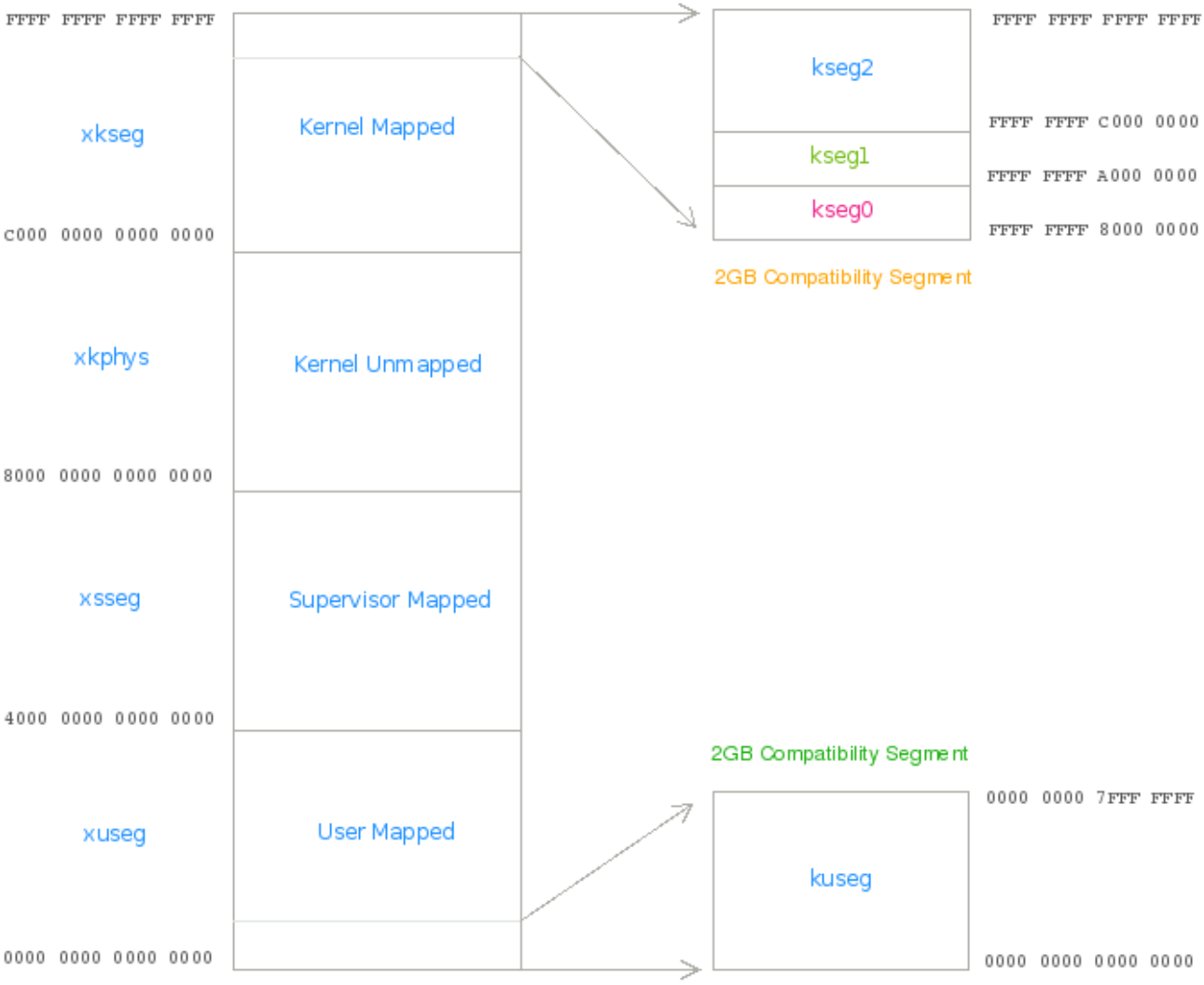

Figure 1.2: 64 bit Virtual Address Space

1.2 物理地址空间

MIPS 平台上，物理地址空间的划分，是由连接 MIPS core 与 RAM、外设之间的桥（有些称为 System/Host Interface 或者 System Controller 抑或 System Bridge）来决定的。

对于 MIPS 平台上的物理地址空间（>= 32 位），工业界长期形成的惯例是将 0x0000 0000 ~ 0x1000 0000 给内存，而将 0x1000 0000 ~ 0x1FFF FFFF 留给 I/O 。若系统的内存大于 256MB，则 256MB 后的内存，其物理地址从 0x2000 0000 开始。如 NEC EMMA3P（使用 36 位物理地址）的物理地址空间划分为：

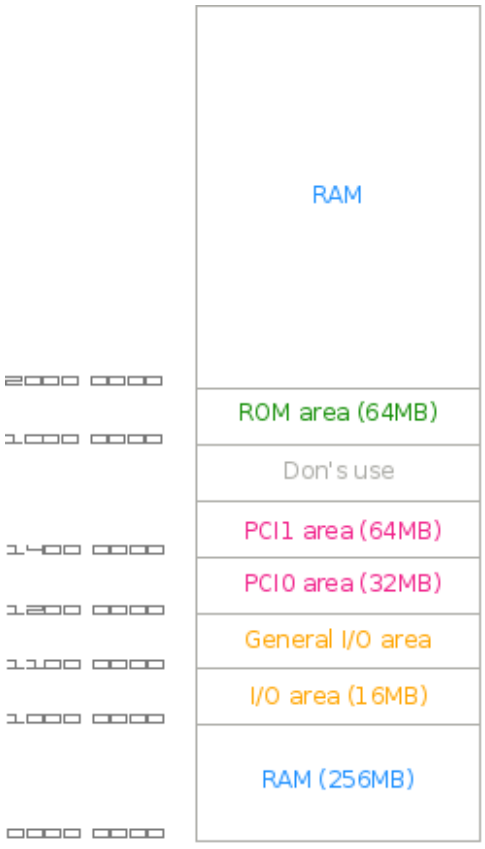

Figure 1.3: EMMA3P's physical address map

MIPS 上，外设的 I/O 寄存器是直接映射到物理地址空间的。

MIPS64 的实现一般都使用超过 32 位的物理地址空间（36 位或 40 位）。

特别地，有些 MIPS32 的实现（只能使用 32 位的虚拟地址空间）却使用超过 32 位的物理地址空间。如 Au1200 使用 36 位的物理地址，则其 Physical Memory 的映射为：

Start Addr	End Addr	Size (MB)	Function
0xF 0000 0000	0xF FFFF FFFF	4096	PCMCIA Interface
0xE 0000 0000	0xD FFFF FFFF	4096	Reserved
0xD 0000 0000	0xD FFFF FFFF	4096	I/O Device
0x1 0000 0000	0xC FFFF FFFF	4096 * 12	Reserved
0x0 F000 0000	0x0 FFFF FFFF	256	Debug Probe
0x0 8000 0000	0x0 EFFF FFFF	1792	Memory (Reserved)
0x0 2000 0000	0x0 7FFF FFFF	1536	Memory
0x0 1800 0000	0x0 1FFF FFFF	128	Flash or ROM
0x0 1400 0000	0x0 17FF FFFF	64	I/O Devices on System Bus
0x0 1200 0000	0x0 13FF FFFF	32	Reserved
0x0 1000 0000	0x0 11FF FFFF	32	I/O Devices on Peripheral Bus
0x0 0000 0000	0x0 0FFF FFFF	256	Memory

Table 1.1: Au1200's Physical Memory Map

对于 MIPS32 使用超过 32 位物理地址的情形，Linux 内核提供了选项 CONFIG_64BIT_PHY_ADDR 支持之，其主要的思想就是将用于存放物理地址的页表项的大小扩展到 64 位。要留意的是 MIPS32 下，访问 512MB 以上的物理地址空间只能通过 TLB 来访问，无法用 kseg0/kseg1 的固定映射去访问的。

一般说来，Bootloader 入口应在 0x1FC0 0000 处，故而此处必为 Flash 或 ROM。

1.3 32 位内核在地址空间上的限制

Linux 在管理物理内存时只使用 kseg0 的前 256MB （后256MB 映射到 I/O），因访问设备的 I/O 寄存器需要立即生效，不能去缓存，故 Linux 操作 I/O 是使用 kseg1 的后 256MB 去访问。除非有特殊的需要，一般内核很少用 kseg1 去访问内存。

当物理内存大于 256MB 时，kseg0 的空间不够用，则需要启用 High Memory 机制，使用 kseg2 的空间，通过 TLB 建立映射去访问。对 64 位情形，不存在这个问题，因为 64 位 Linux 使用 xkphys 区域访问物理内存，cached 和 uncached 区域与整个物理地址空间一样大，足够用矣。

2. TLB 结构和工作方式

2.1 结构

TLB 即 Translation Lookaside Buffer，是为虚拟地址 (VA) 到物理地址 (PA) 转换的硬件机构，其为虚拟存储的硬件基础。TLB 是一个阵列结构，一般有数十项左右，其每项的结构如下图所示：

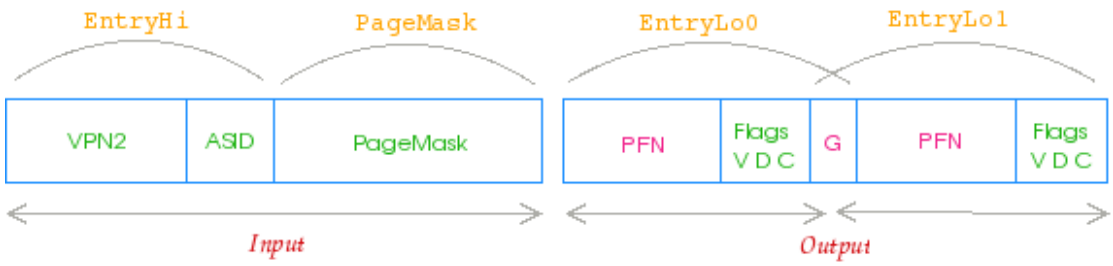

Figure 2.1: TLB Entry Structure

VPN2 为虚页框号，PFN 为物理页框号

可以看到，一个 TLB 项里有两个 PFN，其可以映射以 VPN2 为始框号的两个相邻的偶奇页，其中 VPN2 始终为偶数。则第一个 PFN 对应的虚页框号为 VPN2；第二个 PFN 对应的虚页框号为 VPN2 + 1。

PageMask，页掩码，为支持可变页大小而设。

G，Globle 位，如置该位为 1，则说明该 TLB 项是全局的，可供任何进程使用，TLB 检索时处理器将忽略 ASID 的检查。

此外，针对每个物理页，还有一些控制 Flags：

V: Valid，1 位，置 1 说明该页是有效的。

D: Dirty，1 位，置 1 表明该页中的数据被修改过

C: Cache，3 位，指示对该页所用的缓存算法，可以设置之，而让处理器不将该页数据缓存于 Cache

2.2 工作方式

为了追求性能，MIPS 的实现一般使用全相联的 TLB，则其工作方式为：

处理器取虚址的高位作 VPN，直接将其与 TLB 的所有项同时比较，有匹配的项且该项有效 (V 为 1) 则直接输出 PFN，否则抛出 TLB Refill 异常，后则由 OS 负责在该异常处理里将位于内存的页表读出，并以随机方式写入 TLB，尔后异常返回，重新执行一次访存指令，因此会重走一次 TLB。

为保证某 TLB 项仅供某进程使用，则引入了 ASID，8 位，处理器检索 TLB 时，会以 Entry_Hi 的 ASID 域为当前进程的 ASID 与 TLB 项中的相比较，因此 OS 要小心维护 Entry_Hi 之 ASID 域。

2.3 实例

看一个 MIPS 4KEc 的 TLB 内容，其共有 16 项，可映射 32 个页：

```
Index:  0 pgmask=0x00000000 va=0040c000 asid=58
          [pa=017f4000 c=3 d=0 v=1 g=0]        <---------- vaddr = 0x0040 c000
          [pa=00000000 c=0 d=0 v=0 g=0]        <----------  奇数页没用

Index:  1 pgmask=0x00000000 va=7fe2c000 asid=58
          [pa=09c93000 c=3 d=1 v=1 g=0]
          [pa=00000000 c=0 d=0 v=0 g=0]

Index:  2 pgmask=0x00000000 va=00432000 asid=58
          [pa=01711000 c=3 d=1 v=1 g=0]        <---------- 偶数页 vaddr = 0x0043 2000
          [pa=0170b000 c=3 d=1 v=1 g=0]        <---------- 奇数页 vaddr = 0x0043 3000

Index:  3 pgmask=0x00000000 va=2aaa8000 asid=58
          [pa=0170f000 c=3 d=1 v=1 g=0]
          [pa=01710000 c=3 d=1 v=1 g=0]
```

```
Index:  4 pgmask=0x00000000 va=0041c000 asid=58
            [pa=01727000 c=3 d=0 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  5 pgmask=0x00000000 va=2c3c2000 asid=58
            [pa=01799000 c=3 d=0 v=1 g=0]
            [pa=0179a000 c=3 d=0 v=1 g=0]

Index:  6 pgmask=0x00000000 va=2c320000 asid=58
            [pa=017db000 c=3 d=0 v=1 g=0]
            [pa=017dc000 c=3 d=0 v=1 g=0]

Index:  7 pgmask=0x00000000 va=00444000 asid=58
            [pa=0172f000 c=3 d=0 v=1 g=0]
            [pa=09c9a000 c=3 d=1 v=1 g=0]

Index:  8 pgmask=0x00000000 va=00420000 asid=58
            [pa=0172b000 c=3 d=0 v=1 g=0]
            [pa=0172c000 c=3 d=0 v=1 g=0]

Index:  9 pgmask=0x00000000 va=00434000 asid=58
            [pa=09c90000 c=3 d=1 v=1 g=0]
            [pa=09c98000 c=3 d=1 v=1 g=0]

Index: 10 pgmask=0x00000000 va=00414000 asid=58
            [pa=0171f000 c=3 d=0 v=1 g=0]
            [pa=01720000 c=3 d=0 v=1 g=0]

Index: 11 pgmask=0x00000000 va=2c3b2000 asid=58
            [pa=09c6a000 c=3 d=0 v=1 g=0]
            [pa=09c6b000 c=3 d=0 v=1 g=0]

Index: 12 pgmask=0x00000000 va=0040a000 asid=58
            [pa=00000000 c=0 d=0 v=0 g=0]
            [pa=017f3000 c=3 d=0 v=1 g=0]

Index: 13 pgmask=0x00000000 va=2c368000 asid=58
            [pa=017bb000 c=3 d=0 v=1 g=0]
            [pa=017bc000 c=3 d=0 v=1 g=0]

Index: 14 pgmask=0x00000000 va=00412000 asid=58
            [pa=00000000 c=0 d=0 v=0 g=0]
            [pa=0171e000 c=3 d=0 v=1 g=0]

Index: 15 pgmask=0x00000000 va=00418000 asid=58
            [pa=01723000 c=3 d=0 v=1 g=0]
            [pa=01724000 c=3 d=0 v=1 g=0]
```

注意每项仅有一个 G 位，所以打印出的两个物理页的 g 位是一样的，其来自同一个 G 位。当前使用的是 4KB 大小的页，因此 PageMask 的值都为 0x00000000

3. TLB 控制接口

MIPS 体系结构引入了如下的寄存器和指令作为软硬件的接口，便于软件管理 TLB:

3.1 寄存器

3.1.1 EntryHi

32 位，用于 TLB 读写时存放 VPN2 和 ASID，其结构如下所示：

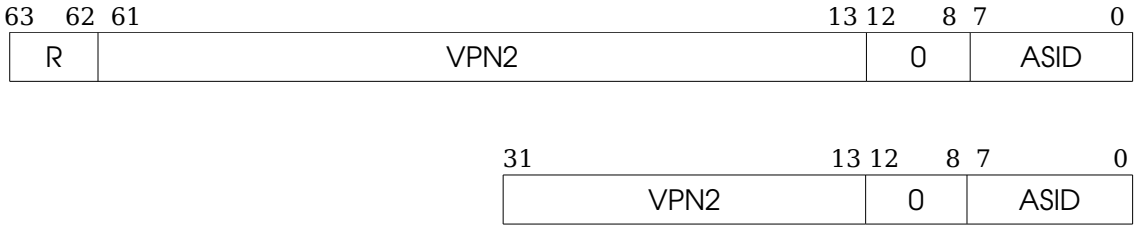

Figure 3.1: EntryHi Structure (MIPS64 and MIPS32)

注意：位于该寄存器的 ASID 还有一个功能，就是这个值被处理器认为是当前进程的 ASID，检索 TLB 时，处理器会以之为当前进程的 ASID 与 TLB 项中的相比较。
另外由于 tlbr 读取 TLB 内容时亦以 EntryHi 存放指定 TLB 项的 VPN2 和 ASID，此会将原 ASID 的内容覆盖，因此在 tlbr 前要将之先保存，尔后要恢复之。

3.1.2 EntryLo0/EntryLo1

32 位，用于 TLB 读写时存放 PFN 和 一些 Flags (C, D, V, G)，其结构如下所示：

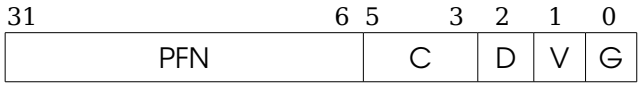

Figure 3.2: EntryLo0/1 Structure (MIPS64 and MIPS32)

EntryLo0 与 EntryLo1 结构一样，只是前者用于偶数页，后者用于奇数页。

3.1.3 PageMask

32 位，用于 TLB 读写时存放 PageMask:

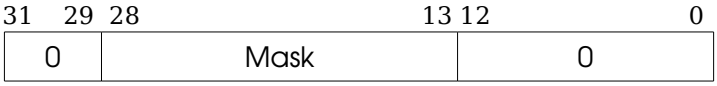

Figure 3.3: PageMask Structure (MIPS64 and MIPS32)

PageMask 的值与页大小之间的对应如下：

PageMask		**Page Size**
Hex	Bin	
0x0000 0000	0000 0000 0000 0000 0000 0000 0000 0000	4KB
0x0000 3000	0000 0000 0000 0000 0011 0000 0000 0000	16KB
0x0000 7000	0000 0000 0000 0000 1111 0000 0000 0000	64KB
0x0003 7000	0000 0000 0000 0011 1111 0000 0000 0000	256KB
0x0007 7000	0000 0000 0000 1111 1111 0000 0000 0000	1MB
0x0037 7000	0000 0000 0011 1111 1111 0000 0000 0000	4MB
0x0077 7000	0000 0000 1111 1111 1111 0000 0000 0000	16MB

以上四个寄存器主要用于和 TLB 项之间的数据交换，其既可作输入数据（写 TLB 项时

用），又可用作存放读出的 TLB 项的内容。

3.1.4 Index

32 位，用作读 TLB (tlbr) 、索引方式写 TLB (tlbwi) 项时指定要被操作的 TLB 入口编号。

3.1.5 Random

32 位，存放一个动态伪随机数，用于指定随机方式写 TLB (tlbwr) 项时，要写的入口。reset 时，该寄存器被硬件置为 TLB 项编号的最大值，运行时不断降低计数，到 Wired 所指示的值后又轮转到最大值，即 Random 的值不会为 0 ~ Wired - 1。
一般情形下，软件无需关心这个寄存器的值。

3.1.6 Wired

32 位，指定要保留的 TLB 项。编号为 0 ~ Wired - 1 的 TLB 项不会被随机写的方式替换。

3.1.7 Contex

32 位，引入的本意是方便异常处理程序读取转换失败的 VPN 以及快速地获取页表的基地址。其结构如下所示：

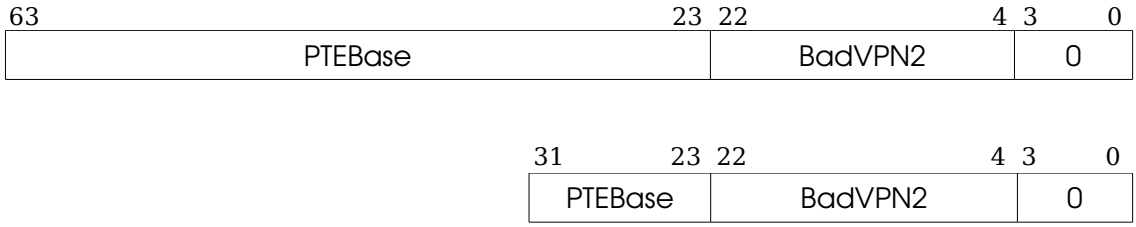

Figure 3.4: Contex Structure (MIPS64 and MIPS32)

TLB Miss 时，硬件会自动将 BadAddr 之高位部分 (VPN) 置入 Context[BadVPN2]。注意硬件不会设置 Context[PTEBase]，更不会自动使用之，这仅仅是体系结构的建议，具体怎么用，取决于 OS 的设计。MIPS Linux 就没有按 MIPS 体系结构的建议使用该域，而是将 smp_processor_id 置入，方便 SMP 对处理器的管理。

3.2 指令

3.2.1 tlbr

TLB 读指令。处理器执行该指令时，直接读取 Index 所指示的 TLB 项，并将其内容分别置入 PageMask，EntryHi，EntryLo0 和 EntryLo1。
其中EntryLo0 之 G 位与 EntryLo1 之 G 位相同，来自 TLB 项中仅有的一个 G 位。

3.2.2 tlbwi/tlbwr

TLB 写指令。
tlbwi 为 TLB Write Indexed，即索引方式写 TLB，入口项由 Index 寄存器指定。
tlbwr 为 TLB Write Random，即随机方式写 TLB，入口项以 Random 寄存器的值为伪随机数，计算相应的入口值。
输入的 TLB 数据，亦置于 EntryHi，EntryLo0 和 EntryLo1。
输入数据时要注意，EntryLo0 之 G 位要与 EntryLo1 之 G 位一致。

3.2.3 tlbp

TLB 查询指令，即 TLB Probe。
其以 EntryHi[VPN2] 检索整个 TLB，有匹配的项则继续检查该项之 ASID 是否与 EntryHi[ASID] 相等，若满足，则将该项的索引值（标号）写入 Index 。若无满足条件的项，则将 Index 的 31 位置为 1。
该指令 probe 的内容仅此而已，若要读取该项的其它数据，可在其后紧随一条 tlbr 。

3.3 示例

3.3.1 读 TLB 的所有项

```
static void dump_tlb_all()
{
    unsigned int pagemask, c0, c1, asid;
    unsigned int entrylo0 = 0, entrylo1 = 0;
    unsigned int entryhi = 0;
    int i = 0;

     /* save current asid */
     asm (
              "mfc0  %0, $10\n\t"
              : "=r" (asid)
          );
     asid &= 0xff;

    printk("\n");
    for (; i <= current_cpu_data.tlbsize - 1; i++) {

          /* read a TLB Entry*/
          asm (
                    "mtc0  %4, $0\n\t"         /* write i to Index */
                    "tlbr          \n\t"       /* read the i tlb entry to register*/
                    "mfc0  %0, $5\n\t"         /* read PageMask */
                    "mfc0  %1, $10\n\t"        /* read EntryHi */
                    "mfc0  %2, $2\n\t"         /* read EntryLo0 */
                    "mfc0  %3, $3\n\t"         /* read EntryLo1 */
                    : "=r" (pagemask), "=r" (entryhi), "=r" (entrylo0), "=r" (entrylo1)
                    : "Jr" (i)
               );

          /* print its content */
          printk("Index: %2d pgmask=0x%08x ", i, pagemask);

          c0 = (entrylo0 >> 3) & 7;
          c1 = (entrylo1 >> 3) & 7;

          printk("va=%08x asid=%02x\n",
                 (entryhi & 0xffffe000), (entryhi & 0xff));
          printk("\t\t\t[pa=%08x c=%d d=%d v=%d g=%d]\n",
                 (entrylo0 << 6) & PAGE_MASK, c0,
                 (entrylo0 & 4) ? 1 : 0,
                 (entrylo0 & 2) ? 1 : 0, (entrylo0 & 1));
          printk("\t\t\t[pa=%08x c=%d d=%d v=%d g=%d]\n",
                 (entrylo1 << 6) & PAGE_MASK, c1,
                 (entrylo1 & 4) ? 1 : 0,
                 (entrylo1 & 2) ? 1 : 0, (entrylo1 & 1));
          printk("\n");
    }

     /* restore asid */
     asm (
              "mtc0  %0, $10\n\t"
              : : "Jr" (asid)
          );
}
```

注意 EntryLo0[PFN] 是物理地址的 31 ~ 12 位，现在其置于 EntryLo 的 25 ~ 6 位，

因此为获得物理地址则要将 entrylo0 的值左移 6 位并将低 12 位置 0 即可。

其在 VR5500 上的输出为：

```
Index:  0 pgmask=0x00000000 va=2aac2000 asid=39
            [pa=016b8680 c=3 d=0 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  1 pgmask=0x00000000 va=2ac66000 asid=38
            [pa=086a5780 c=3 d=1 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  2 pgmask=0x00000000 va=2ac78000 asid=38
            [pa=016ae680 c=3 d=0 v=1 g=0]
            [pa=016af680 c=3 d=0 v=1 g=0]

Index:  3 pgmask=0x00000000 va=2acf6000 asid=38
            [pa=017d0680 c=3 d=0 v=1 g=0]
            [pa=017d1680 c=3 d=0 v=1 g=0]

Index:  4 pgmask=0x00000000 va=2ac00000 asid=38
            [pa=09784680 c=3 d=0 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  5 pgmask=0x00000000 va=2aab4000 asid=38
            [pa=0928c680 c=3 d=0 v=1 g=0]
            [pa=0928d680 c=3 d=0 v=1 g=0]

Index:  6 pgmask=0x00000000 va=5555e000 asid=38
            [pa=00000000 c=0 d=0 v=0 g=0]
            [pa=08436680 c=3 d=0 v=1 g=0]

Index:  7 pgmask=0x00000000 va=2abac000 asid=38
            [pa=095cc680 c=3 d=0 v=1 g=0]
            [pa=09661680 c=3 d=0 v=1 g=0]
......  ......
......  ......
```

该 VR5500 之 TLB 有 64 项，为节约篇幅这里仅列出前 8 项。

3.3.2 写一项入 TLB

```
static void write_tlb_entry()
{
    unsigned int pagemask = 0, asid = 0;
    unsigned int entrylo0 = (0x00543 << 6) + (3 << 3) + (0 << 2) + (0 << 1) + 1;
    unsigned int entrylo1 = (0x00777 << 6) + (3 << 3) + (0 << 2) + (0 << 1) + 1;
    unsigned int entryhi = 0x00abc000 + 0xa;
    int i = 0;

    /* save current asid */
    asm ( "mfc0   %0, $10\n\t" : "=r" (asid) );
    asid &= 0xff;

    /* random write 5 tlb entry */
    for(; i < 5; i++)
    {
        asm (
                "mtc0  %0, $5\n\t"         /* write PageMask */
                "mtc0  %1, $10\n\t"       /* write EntryHi */
                "mtc0  %2, $2\n\t"         /* write EntryLo0 */
                "mtc0  %3, $3\n\t"         /* write EntryLo1 */
                "tlbwr       \n\t"             /* TLB random write */
                : : "r" (pagemask), "r" (entryhi), "r" (entrylo0), "r" (entrylo1), "Jr" (i)
            );
    }

    asm (
            "mfc0  %0, $5\n\t"          /* read PageMask */
            "mfc0  %1, $10\n\t"   /* read EntryHi */
            "mfc0  %2, $2\n\t"          /* read EntryLo0 */
            "mfc0  %3, $3\n\t"          /* read EntryLo1 */
            : "=r" (pagemask), "=r" (entryhi), "=r" (entrylo0), "=r" (entrylo1)
        );

    printk("pgmsk = 0x%08x, entryhi = 0x%08x, entrylo0 = 0x%08x, entrylo1 = 0x%08x\n",
            pagemask, entryhi, entrylo0, entrylo1);

    /* restore asid */
    asm ( "mtc0   %0, $10\n\t" : : "Jr" (asid) );
}
```

上面的函数随机写入 5 项。

注意：执行后，查看时，可能只看到一到两项，因为这期间可能被后入的其它项所替换，下一节的实现（写入一个固定项）则不会有这个问题。

VR5500 上，执行前 TLB 内没有 va 为 0x00abc000 的项。

执行后：

```
......  ......
......  ......
Index: 24 pgmask=4kb va=00408000 asid=3f
        [pa=0165e000 c=3 d=0 v=1 g=0]
        [pa=0165f000 c=3 d=0 v=1 g=0]

Index: 25 pgmask=4kb va=2f43e000 asid=b9
        [pa=0169f000 c=3 d=0 v=1 g=0]
        [pa=08494000 c=3 d=1 v=1 g=0]
```

```
Index: 26 pgmask=4kb va=00abc000 asid=0a
        [pa=00543000 c=3 d=0 v=0 g=1]
        [pa=00777000 c=3 d=0 v=0 g=1]

Index: 27 pgmask=4kb va=2f440000 asid=3f
        [pa=08565000 c=3 d=1 v=1 g=0]
        [pa=0047c000 c=3 d=1 v=1 g=0]

Index: 28 pgmask=4kb va=004e4000 asid=b9
        [pa=08497000 c=3 d=1 v=1 g=0]
        [pa=00000000 c=0 d=0 v=0 g=0]

Index: 29 pgmask=4kb va=00abc000 asid=0a
        [pa=00543000 c=3 d=0 v=0 g=1]
        [pa=00777000 c=3 d=0 v=0 g=1]

Index: 30 pgmask=4kb va=00418000 asid=b9
        [pa=017ff000 c=3 d=0 v=1 g=0]
        [pa=00600000 c=3 d=0 v=1 g=0]

Index: 31 pgmask=4kb va=2f41c000 asid=b9
        [pa=01729000 c=3 d=0 v=1 g=0]
        [pa=00000000 c=0 d=0 v=0 g=0]
......  ......
......  ......
```

3.3.3 写两个固定项入 TLB

```
static void write_wired_tlb_entry()
{
    unsigned int pagemask = 0, asid = 0;
    unsigned int entrylo0 = (0x00543 << 6) + (3 << 3) + (0 << 2) + (0 << 1) + 1;
    unsigned int entrylo1 = (0x00777 << 6) + (3 << 3) + (0 << 2) + (0 << 1) + 1;
    unsigned int entryhi = 0x00abc000 + 0xa;
     int i = 3;

    /* save current asid */
    asm ( "mfc0   %0, $10\n\t" : "=r" (asid) );
    asid &= 0xff;

    asm (
            "mtc0  %0, $5\n\t"        /* write PageMask */
            "mtc0  %1, $10\n\t"      /* write EntryHi */
            "mtc0  %2, $2\n\t"        /* write EntryLo0 */
            "mtc0  %3, $3\n\t"        /* write EntryLo1 */

            "mfc0  $4, $6\n\t"        /* read Wired */
            "addi  $3, $4, 2\n\t"
            "mtc0  $3, $6\n\t"        /* wired += 2 */

            "mtc0  $4, $0\n\t"        /* write wired to Index */
            "tlbwi       \n\t"        /* TLB indexed write */

            "addi  $4, $4, 1\n\t"
            "mtc0  $4, $0\n\t"        /* write wired + 1 to Index */
            "tlbwi       \n\t"       /* TLB indexed write */

            : : "r" (pagemask), "r" (entryhi), "r" (entrylo0), "r" (entrylo1)
        );

    asm (
            "mfc0  %0, $5\n\t"        /* read PageMask */
            "mfc0  %1, $10\n\t"       /* read EntryHi */
            "mfc0  %2, $2\n\t"        /* read EntryLo0 */
            "mfc0  %3, $3\n\t"        /* read EntryLo1 */
            : "=r" (pagemask), "=r" (entryhi), "=r" (entrylo0), "=r" (entrylo1)
        );

    printk("pgmsk = 0x%08x, entryhi = 0x%08x, entrylo0 = 0x%08x, entrylo1 = 0x%08x\n",
            pagemask, entryhi, entrylo0, entrylo1);

    /* restore asid */
    asm ( "mtc0   %0, $10\n\t" : : "Jr" (asid) );
}
```

该函数每执行一次则将 Wired += 2，即分配 2 项，用作固定映射。如执行前 TLB 为：

```
Index:  0 pgmask=4kb va=004a2000 asid=b9
        [pa=084ad000 c=3 d=0 v=1 g=0]
        [pa=084ae000 c=3 d=0 v=1 g=0]

Index:  1 pgmask=4kb va=c0188000 asid=b9
        [pa=0998b000 c=3 d=1 v=1 g=1]
        [pa=09a26000 c=3 d=1 v=1 g=1]

Index:  2 pgmask=4kb va=2f3ba000 asid=3f
        [pa=016f9000 c=3 d=0 v=1 g=0]
        [pa=016fa000 c=3 d=0 v=1 g=0]
```

```
Index:  3 pgmask=4kb va=0046a000 asid=b9
        [pa=00664000 c=3 d=0 v=1 g=0]
        [pa=00000000 c=0 d=0 v=0 g=0]

Index:  4 pgmask=4kb va=004de000 asid=b9
        [pa=017dc000 c=3 d=0 v=1 g=0]
        [pa=08bef000 c=3 d=1 v=1 g=0]
......  ......
......  ......
```

执行一次后，TLB 为：

```
Index:  0 pgmask=4kb va=00abc000 asid=0a
          [pa=00543000 c=3 d=0 v=0 g=1]
          [pa=00777000 c=3 d=0 v=0 g=1]

Index:  1 pgmask=4kb va=00abc000 asid=0a
          [pa=00543000 c=3 d=0 v=0 g=1]
          [pa=00777000 c=3 d=0 v=0 g=1]

Index:  2 pgmask=4kb va=2f3ba000 asid=3f
        [pa=016f9000 c=3 d=0 v=1 g=0]
        [pa=016fa000 c=3 d=0 v=1 g=0]

Index:  3 pgmask=4kb va=0046a000 asid=b9
        [pa=00664000 c=3 d=0 v=1 g=0]
        [pa=00000000 c=0 d=0 v=0 g=0]

Index:  4 pgmask=4kb va=004de000 asid=b9
        [pa=017dc000 c=3 d=0 v=1 g=0]
        [pa=08bef000 c=3 d=1 v=1 g=0]

......  ......
......  ......
```

3.3.4 TLB 查询

下面的代码查询一个 VA 在 TLB 有无匹配的项：

```
static void probe_tlb()
{
   unsigned int entryhi = 0x00abc000 + 0xa;
    unsigned int index = 0x55;
    unsigned int asid = 0;

    /* save current asid */
    asm ("mfc0    %0, $10\n\t" : "=r" (asid));
    asid &= 0xff;

    asm ( "mfc0   %0, $0\n\t" : "=r" (index));
    printk("Previous Index = 0x%08x\n", index);

    asm (
            "mtc0  %0, $10\n\t"   /* write EntryHi */
            "tlbp   \n\t"             /* probe TLB entry*/
            : : "Jr" (entryhi)
        );

    printk("Probe EntryHi: 0x%08x\n", entryhi);

    asm ( "mfc0   %0, $0\n\t" : "=r" (index));
    printk("After TLB probe, Index = 0x%08x\n", index);

    /* restore asid */
    asm ( "mtc0   %0, $10\n\t" : : "Jr" (asid));
}
```

执行 write_wired_tlb_entry() 后，再执行 probe_tlb() 其输出为：

```
Previous Index = 0x80000000
Probe EntryHi: 0x00abc00a
After TLB probe, Index = 0x00000000
```

查到 TLB 的第一项 (Index 0) 就匹配 va = 0x00abc000，因该项 g = 1 所以忽略 asid 的检查。可以看到 TLB probe 时，若有多个项匹配则取最前的一个；而 TLB 正常工作时，如有多项命中则会直接抛出异常。

以上所有函数可以借助 Linux 的 SysRq 机制来直接执行，可以从该处获取修改后的 sysrq.c 文件：http://people.openrays.org/~comcat/

4. TLB 异常

MIPS 引入以下三个异常来支持软件维护 TLB。

由于 TLB 的容量有限，而一个 OS 的页表众多，这个就需要软件在适当的时候填充 TLB，这个是引入 TLB 重填异常的目的。

页表每项有一个 V 位，用于指示该项是否有效，引入该位的目的意在支持软件 flush TLB（直接置该位为 0）。硬件在匹配时检测到一个匹配项无效时，相应的就需要软件的处理，这个是引入 TLB 无效异常的目的。

4.1 TLB Refill exception

TLB 重填异常，当所访问的虚拟地址在 TLB 中没有对应项时，处理器会抛出该异常。

在抛出异常前，硬件会自动设置如下寄存器：

BadVaddr: The failing address
Context: The BadVPN2 field contains VA[31..13] of the failing address
EntryHi: The VPN2 field contains VA[31..13] of the failing address;
CAUSE[ExcCode]: TLBL(2), TLBS(3)

而该异常处理的入口会因 STATUS 寄存器的 EXL 位的不同而不同：

EXL = 0, vec_addr = 0x8000 0000
EXL = 1, vec_addr = 0x8000 0180 (General exception)

OS 在该异常的处理函数中将转换失败的 VA 所对应的页表项填入 TLB（随机替换掉一项），后异常返回，又重访问一次该 VA，则可得其所对应之 PA。

4.2 TLB Invalid exception

TLB 无效异常。处理器用 VA 匹配 TLB 时，有命中但该项之有效位 V 为 0 则引发给异常。

在抛出异常前，硬件会自动设置如下寄存器：

BadVaddr: The failing address
Context: The BadVPN2 field contains VA[31..13] of the failing address
EntryHi: The VPN2 field contains VA[31..13] of the failing address;
ExeCode: TLBL(2), TLBS(3)

该异常处理的入口则为：

vec_addr = 0x8000 0180 (General exception)

Use tlbp to distinguish TLB Invalid or TLB Refill

4.3 TLB Modified Exception

CAUSE:
A store reference to a mapped address when the matching TLB entry is valid, but the entry's D bit is zero, indicating that the page is not writable.

在抛出异常前，硬件会自动设置如下寄存器：

BadVaddr: The failing address
Context: The BadVPN2 field contains VA[31..13] of the failing address
EntryHi: The VPN2 field contains VA[31..13] of the failing address;
ExeCode: Mod(1)

Vector:

vec_addr = 0x8000 0180 (General exception)

5. 内核代码分析

本章仅概要分析一下与体系结构紧密相关的代码。

5.1 页表管理

5.1.1 多级页表基本思想

最初引入多级分页模型是为了在虚拟内存下让页表也能交换出内存（早期的内存很小）。对 32 位地址空间、4KB 的页大小、页表的每项大小为 4B 的情形，其所需的页表空间为：

```
4GB/4KB * 4 = 2^22
```

即 4MB，这个在早期还是一个相当可观的内存消耗，为了能在内存不足的情形下也能将页表部分对换出来，因此就对页表进行分页，因每页可容纳的页表项为 2^10，则共需 1024 页来容纳 4MB 的页表，很自然的就需要一张表来索引这 1024 个页表，这个实际上就是 PGD。

引入多级页表的另一个原因就是可以按需分配页表所需的空间。一级页表的情形，每创建一个进程，为保证页表项的连续，都要为其一次分配 4MB (4GB/4KB * 4) 的页表，尽管其只用到很少的一部分。多级页表下，每创建一个进程，只要为其一次分配 PGD 所需的空间即可（下一级则在其需要时再分配），PGD 等于是将原来连续的空间划分成一个个子空间，只要保证 PGD 项之间的连续即可了。

对于 32 位内核（包括 MIPS32 的实现运行于 32 位模式，以及 MIPS64 的实现兼容运行于 32 位模式下），MIPS Linux 使用两级分页，即页全局目录 (PGD) 和页表 (PT)。对于 64 位内核（MIPS64 运行于 64 位模式下），MIPS Linux 使用三级分页，即页全局目录 (PGD)，页中间目录 (PMD) 和页表。

尽管从 2.6.11 始，内核就支持四级分页模型（PGD, PUD, PMD, PT），但 MIPS 用到

的也就是两级和三级。当其使用三级分页时，其 PUD 是为空的，两级分页时其 PUD、PMD 都为空，内核引入 include/asm-generic/pgtable-nopud.h 和 include/asm-generic/pgtable-nopmd.h 来分别支持不使用 PUD 和 PMD 的情形。

PT 的每项内容就是该页对应的物理地址（页框号）和一些控制位，而 PGD, PUD, PMD 的每项内容实际就是下级表（大小一页）的基地址。

5.1.2 多级页表下虚拟地址的划分

在多级分页模型下，PUD, PMD, PT，其每张表的容量为一个页大小（64 位 Linux 的 PGD 占用两个页），PGD 因为在最上层，则可根据页大小和虚拟地址空间大小灵活地调整容量。

5.1.2.1 32 位情形

知道了多级页表的基本思想，32 位下 2 级页表虚拟地址的划分就会很清晰了：

对 4KB 的页大小，VA[11:0] 为页内偏移与物理地址一致
因每页 (PT) 的页表项为 2^10 (4KB/4B)，则 VA[21:12] 用来索引 PT
因 PGD 也有 2^10 项，则最高的 10 位 VA[31:22] 用来索引 PGD

对 16KB 的页大小，VA[13:0] 为页内偏移与物理地址一致
因每页 (PT) 的页表项为 2^12 (16KB/4B)，则 VA[25:14] 共 12 位用之索引 PT
故 VA 余下的 6 位用之索引 PGD，因此 PGD 只要有 64 项即可，即每个 PGD 的项可映射 2^12 * 2^14 = 2^26 的空间。

则其工作方式如下图所示：

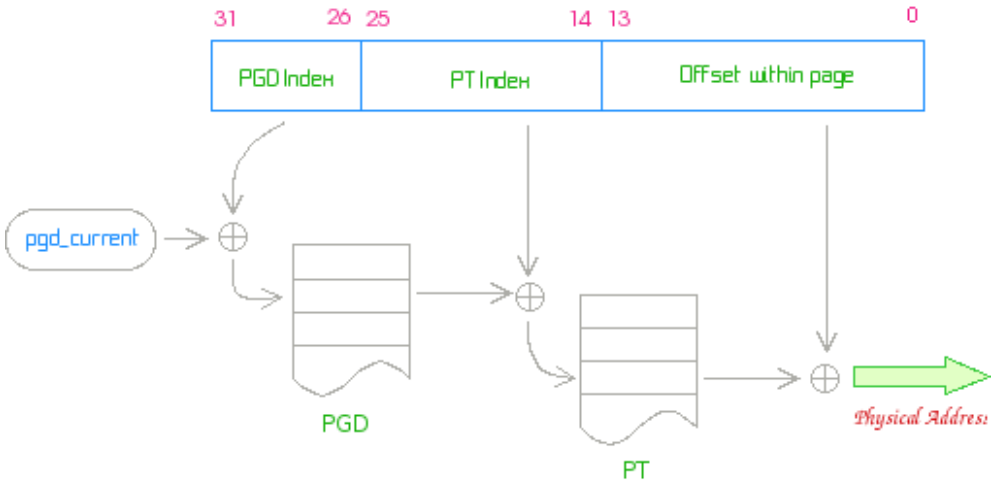

Figure 5.1: 16KB page size, 32 bit virtual address split

pgd_current 用于存放当前进程之 PGD 的基址，其定义于 [arch/mips/mm/init.c]:

```
unsigned long pgd_current[NR_CPUS];
```

可以看到：SMP 下，每个 CPU 都有一个 pgd_current。

对于 32 位的内核运行在使用大于 32 位物理地址的硬件上的情形（内核选项 CONFIG_64BIT_PHY_ADDR，主要是 MIPS32 的实现使用大于 32位的物理地址；还有就是 MIPS64 的实现其使用超过 32 位的物理地址，且其兼容运行于 MIPS32 模式下）为了支持大于 32 位的物理地址，页表的每项大小要扩展为 8 字节，则每张页表可容纳的项数为：PAGE_SIZE/8 。扩展的目的旨在容纳大于 32 位的物理地址，其虚拟地址的位数仍然是 32，则所用容纳虚拟地址的项的大小无需变化，即 PGD 的项大小仍然是 4 字节。则 16KB 的页大小，其 32 位虚拟地址的划分为：

VA[13:0]	页内偏移
VA[24:14]	用于索引页表 （2048 项）
VA[31:25]	用于索引 PGD （128 项）

在此情形下页表的容量减为原来的一半，但 PGD 的容量却是原来的 2 倍，则其依然能映射 4GB 的虚拟地址空间大小。

5.1.2.2 64 位情形

对 64 位的情形，因使用长于 32 位的虚拟地址，且都使用大于 32位的物理地址，因此 PGD, PMD, PT 的每项大小都要相应地扩展到 8 字节，计算基本思想不变。如 4KB 的大小，其虚拟地址的划分为：

VA[11:0]	页内偏移
VA[20:12]	用于索引页表 （页表大小为一页，512 项，项大小为 8 字节）
VA[29:21]	用于索引 PMD （PMD 大小为一页，512 项，项大小为 8 字节）
VA[39:30]	用于索引 PGD （PGD 大小为两页，1024 项，项大小为 8 字节）

8 KB 页大小下，虚拟地址的划分为：

VA[12:0]	页内偏移
VA[22:13]	用于索引页表 （页表大小为一页，1024 项，项大小为 8 字节）
VA[32:23]	用于索引 PMD （PMD 大小为一页，1024 项，项大小为 8 字节）
VA[42:33]	用于索引 PGD （PGD 大小为一页，1024 项，项大小为 8 字节）

16 KB 下，虚拟地址的划分为：

VA[13:0]	页内偏移
VA[24:14]	用于索引页表 （页表大小为一页，2048 项，项大小为 8 字节）
VA[35:25]	用于索引 PMD （PMD 大小为一页，2048 项，项大小为 8 字节）
VA[46:36]	用于索引 PGD （PGD 大小为一页，2048项，项大小为 8 字节）

64 KB 页大小下计算方法类似，且 PGD 也仅有一页。

至于为何唯独 4 KB 时 PGD 的大小为两页，这个是由常量 PTRS_PER_PGD 来决定的：

[include/asm-mips/pgtable-64.h]:

```
#ifdef CONFIG_PAGE_SIZE_4KB
#define PGD_ORDER       1
#define PUD_ORDER       aieeee_attempt_to_allocate_pud
#define PMD_ORDER       0
#define PTE_ORDER       0
#endif
#ifdef CONFIG_PAGE_SIZE_8KB
#define PGD_ORDER       0
#define PUD_ORDER       aieeee_attempt_to_allocate_pud
#define PMD_ORDER       0
#define PTE_ORDER       0
#endif
#ifdef CONFIG_PAGE_SIZE_16KB
#define PGD_ORDER       0
#define PUD_ORDER       aieeee_attempt_to_allocate_pud
#define PMD_ORDER       0
#define PTE_ORDER       0
```

```
#endif
#ifdef CONFIG_PAGE_SIZE_64KB
#define PGD_ORDER       0
#define PUD_ORDER       aieeee_attempt_to_allocate_pud
#define PMD_ORDER       0
#define PTE_ORDER       0
#endif

#define PTRS_PER_PGD    ((PAGE_SIZE << PGD_ORDER) / sizeof(pgd_t))
```

很显然在 4KB 时，因为 PGD_ORDER 为 1，使得 PTRS_PER_PGD 为 1024；其它页大小时，因为 PGD_ORDER 皆为 0，则只用一页，即大小为 PAGE_SIZE。

可以看到，当前 64 位 Linux 的设计，在不同的页大小下，其使用的虚拟地址空间大小是不一样的，即 4KB 时为 40 位；8KB 时为 43 位；16KB 时为 47 位；64KB 时为 55 位。这也是一个无奈之举，因为内核在编译时就需要确定 PGD 的大小，这时是不能去动态的获取实际实现的虚拟地址长度（获取 EntryHi 之 VPN2 域的有效位），只能估测稍大点，便于实现吧，尽管可能尾部的 PGD 项根本用不上，浪费一些空间。

5.1.3 页表的结构定义

在第一个线程 init 生成前，内核静态定义了一个页目录表 (PGD) 位于 [arch/mips/mm/init.c]:

```
#define __page_aligned(order) __attribute__((__aligned__(PAGE_SIZE<<order)))
pgd_t swapper_pg_dir[PTRS_PER_PGD] __page_aligned(PGD_ORDER);
```

PTRS_PER_PGD 为一个 PGD 所应含有的项数，其定义于 [include/asm-mips/pgtable-32.h]:

```
#define PTRS_PER_PGD    ((PAGE_SIZE << PGD_ORDER) / sizeof(pgd_t))
```

32 位时，PTRS_PER_PGD 通常为 USER_PTRS_PER_PGD 的 2 倍。上面的定义是有

问题的。最好能修正为：0x1 0000 0000ULL / PGDIR_SIZE 。

USER_PTRS_PER_PGD 定义在 [include/asm-mips/pgtable-32.h]:

```
#define USER_PTRS_PER_PGD   (0x80000000UL/PGDIR_SIZE)
```

64 bit 时为 [include/asm-mips/pgtable-64.h]：

```
#define USER_PTRS_PER_PGD   (TASK_SIZE / PGDIR_SIZE)
```

64 bit 时 TASK_SIZE 为 1TB

PGD_ORDER 在最 2.6.27 中定义为[include/asm-mips/pgtable-32.h]:

```
#define __PGD_ORDER (32 - 3 * PAGE_SHIFT + PGD_T_LOG2 + PTE_T_LOG2)
#define PGD_ORDER   (__PGD_ORDER >= 0 ? __PGD_ORDER : 0)
```

PGD_T_LOG2, PTE_T_LOG2 分别为 PGD 每项的字节大小取 Log2 和 PT 每项的字节大小取 Log2 ， 他们定义于 [include/asm-mips/pgtable.h]:

```
#define PGD_T_LOG2  ffz(~sizeof(pgd_t))
#define PTE_T_LOG2  ffz(~sizeof(pte_t))
```

ffz () 宏用于找寻一个字内从右到左第一个为 0 的位的编号，如 ffz (1) = 1, ffz (0x3) = 2，pgd_t 大小为 4 时 ffz (~0x4) = 2，即为 Log2 (4) 的值。pgd_t，pte_t 分别为 PGD 和 PT 的每项类型，定义于 [include/asm-mips/page.h]:

```
typedef struct { unsigned long pgd; } pgd_t;
```

pte_t 的定义考虑的情形稍多：

```
#ifdef CONFIG_64BIT_PHYS_ADDR
 #ifdef CONFIG_CPU_MIPS32
  typedef struct { unsigned long pte_low, pte_high; } pte_t;
```

```
  #define pte_val(x) ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
  #define __pte(x) ({ pte_t __pte = {(x), ((unsigned long long)(x)) >> 32};
__pte; })
 #else
  typedef struct { unsigned long long pte; } pte_t;
  #define pte_val(x) ((x).pte)
  #define __pte(x)   ((pte_t) { (x) } )
 #endif
#else
 typedef struct { unsigned long pte; } pte_t;
 #define pte_val(x)  ((x).pte)
 #define __pte(x)    ((pte_t) { (x) } )
#endif
```

只要其使用超过 32 位的物理地址（CONFIG_64BIT_PHY_ADDR，MIPS 64 的实现使用超过 32 位的物理地址，但其兼容运行于 MIPS 32 模式下。实际情形下，一般的 MIPS 64 的实现也就使用 40 位的物理地址），则其 pte_t 就是 8 字节。

因此，32 位下 PGD_ORDER 始终为 0；只有在 CONFIG_64BIT_PHY_ADDR =y 时，PGD_ORDER 才有可能为 1 （只在 4KB 页大小时），因为此时 PTE_T_LOG2 为 3（PGD_T_LOG2 仍然为 2）。

可以将 PGD_ORDER 理解为这样一个参数，其在每页能容纳的页表项数减小时，用于调节 PGD 的项数（增加 PGD ）。例如当 pte_t 为 8 字节时，页表的容量是原来的一半，则相应的 PGD 的容量要扩充为原来的 2 倍。

5.1.4 PGD 的初始化

系统最初的 PGD 定义为 swapper_pg_dir，系统的第一个线程 init 即使用该 PGD，对其的初始化工作位于[arch/mips/mm/pgtable-32.h]：

```
void __init pagetable_init(void)
{
```

```
    unsigned long vaddr;
    pgd_t *pgd_base;
#ifdef CONFIG_HIGHMEM
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;
#endif

    /* Initialize the entire pgd.  */
    pgd_init((unsigned long)swapper_pg_dir);
    pgd_init((unsigned long)swapper_pg_dir
         + sizeof(pgd_t) * USER_PTRS_PER_PGD);

    pgd_base = swapper_pg_dir;

    /*
     * Fixed mappings:
     */
    vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
    fixrange_init(vaddr, 0, pgd_base);

#ifdef CONFIG_HIGHMEM
    /*
     * Permanent kmaps:
     */
    vaddr = PKMAP_BASE;
    fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);

    pgd = swapper_pg_dir + __pgd_offset(vaddr);
    pud = pud_offset(pgd, vaddr);
    pmd = pmd_offset(pud, vaddr);
    pte = pte_offset_kernel(pmd, vaddr);
    pkmap_page_table = pte;
#endif
}

void pgd_init(unsigned long page)
{
```

```
    unsigned long *p = (unsigned long *) page;
    int i;

    for (i = 0; i < USER_PTRS_PER_PGD; i+=8) {
        p[i + 0] = (unsigned long) invalid_pte_table;
        p[i + 1] = (unsigned long) invalid_pte_table;
        p[i + 2] = (unsigned long) invalid_pte_table;
        p[i + 3] = (unsigned long) invalid_pte_table;
        p[i + 4] = (unsigned long) invalid_pte_table;
        p[i + 5] = (unsigned long) invalid_pte_table;
        p[i + 6] = (unsigned long) invalid_pte_table;
        p[i + 7] = (unsigned long) invalid_pte_table;
    }
}
```

HIGHMEM 的情形不关心，略过。
可以看到每调用一次 pgd_init() 即初始化 USER_PTRS_PER_PGD 项，两次调用正好初始化完整个 PGD（PTRS_PER_PGD 为 USER_PTRS_PER_PGD 的两倍）。
在 pgd_init() 中主要将 PGD 的每项指向 invalid_pte_table。这个则定义于 [arch/mips/mm/init.c]：

```
pte_t invalid_pte_table[PTRS_PER_PTE] __page_aligned(PTE_ORDER);
```

invalid_pte_table 即为一个页表，是一个全局量，不会进入实际使用，只是为 PGD 提供一个可以参考的入口，判断时只要 PGD 的项指向 invalid_pte_table 即认为此 PGD 项为无效。

init 起来后，往后的进程之 PGD，系统都是在 fork() 时调用 pgd_alloc() 获取之：

[include/asm-mips/pgalloc.h]

```
static inline pgd_t *pgd_alloc(struct mm_struct *mm)
{
    pgd_t *ret, *init;

    ret = (pgd_t *) __get_free_pages(GFP_KERNEL, PGD_ORDER);
```

```
    if (ret) {
        init = pgd_offset(&init_mm, 0UL);
        pgd_init((unsigned long)ret);
        memcpy(ret + USER_PTRS_PER_PGD, init + USER_PTRS_PER_PGD,
               (PTRS_PER_PGD - USER_PTRS_PER_PGD) * sizeof(pgd_t));
    }

    return ret;
}
```

可以看到，init 以后的进程之 PGD 初始化，首先调用 pgd_init() 初始化前 USER_PTRS_PER_PGD 项，这些项对应 2G 的用户空间（32bit 下），即将用户空间的 PGD 项指向 invalid_pte_table，留待后来加载进程代码镜像时，通过系统的缺页异常填充之。

而后，内核空间的 PGD 项则拷贝自 init_mm （属于 init 进程），即 init 的子进程（或线程）与 init 共享内核空间。

5.1.5 页表项的控制位

页表每项大小为 32 位或 64 位，不论其大小为几何，每项的低位都用于描述该页的属性，具体的位偏移定义在 [include/asm-mips/pgtable-bits.h]:

```
#if defined(CONFIG_64BIT_PHYS_ADDR) && defined(CONFIG_CPU_MIPS32)

#define _PAGE_PRESENT               (1<<6)  /* implemented in software */
#define _PAGE_READ                  (1<<7)  /* implemented in software */
#define _PAGE_WRITE                 (1<<8)  /* implemented in software */
#define _PAGE_ACCESSED              (1<<9)  /* implemented in software */
#define _PAGE_MODIFIED              (1<<10) /* implemented in software */
#define _PAGE_FILE                  (1<<10) /* set:pagecache unset:swap */

#define _PAGE_R4KBUG                (1<<0)  /* workaround for r4k bug  */
#define _PAGE_GLOBAL                (1<<0)
```

```
#define _PAGE_VALID                (1<<1)
#define _PAGE_SILENT_READ          (1<<1)  /* synonym                  */
#define _PAGE_DIRTY                (1<<2)  /* The MIPS dirty bit       */
#define _PAGE_SILENT_WRITE         (1<<2)
#define _CACHE_SHIFT               3
#define _CACHE_MASK                (7<<3)

#else

#define _PAGE_PRESENT              (1<<0)  /* implemented in software */
#define _PAGE_READ                 (1<<1)  /* implemented in software */
#define _PAGE_WRITE                (1<<2)  /* implemented in software */
#define _PAGE_ACCESSED             (1<<3)  /* implemented in software */
#define _PAGE_MODIFIED             (1<<4)  /* implemented in software */
#define _PAGE_FILE                 (1<<4)  /* set:pagecache unset:swap */

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

#define _PAGE_GLOBAL               (1<<8)
#define _PAGE_VALID                (1<<9)
#define _PAGE_SILENT_READ          (1<<9)  /* synonym                  */
#define _PAGE_DIRTY                (1<<10) /* The MIPS dirty bit       */
#define _PAGE_SILENT_WRITE         (1<<10)
#define _CACHE_UNCACHED            (1<<11)
#define _CACHE_MASK                (1<<11)

#else

#define _PAGE_R4KBUG               (1<<5)  /* workaround for r4k bug  */
#define _PAGE_GLOBAL               (1<<6)
#define _PAGE_VALID                (1<<7)
#define _PAGE_SILENT_READ          (1<<7)  /* synonym                  */
#define _PAGE_DIRTY                (1<<8)  /* The MIPS dirty bit       */
#define _PAGE_SILENT_WRITE         (1<<8)
#define _CACHE_SHIFT               9
#define _CACHE_MASK                (7<<9)

#endif
#endif /* defined(CONFIG_64BIT_PHYS_ADDR && defined(CONFIG_CPU_MIPS32) */
```

仅考虑非 64 位物理地址且不是 R3000 和 TX39XX 的情形（彩色代码区），则其使用每项的低 12 位用作控制域 (pgprot)，其中 PTE[5:0] 是软件实现，PTE[11:6] 则正好对应于 TLB 每项的 EntryLo0/1 的低 6 位控制域，分别为 3 位的 Cache 一致性属性，1 位 Dirty 位，1 位有效位以及 1 位全局位。其中内核分别为 DIRTY 位和 VALID 位给了一个别名：_PAGE_SILENT_WRITE，_PAGE_SILENT_READ。在此情形下，当 TLB Miss 异常出现，内核在处理时，读取到 PTE 后，直接将 PTE 右移 6 位即得 EntryLo0/1 的值。

5.1.6 操作页表的一些辅助宏和函数

这些基本定义在 [include/asm-mips/pgtable.h]，以下仅择其要者，概述之:

`mk_pte(struct page *page, pgprot_t pgprot)`

将 page 转为 pfn 与页状态位 pgprot 合并得一个 PTE

`set_pte(pte_t *ptep, pte_t pte_dat)`

将 pte_dat 的值置入 ptep 指向的 PTE

`pte_clear(struct mm_struct *mm, unsigned long addr, pte_t *ptep)`

将 ptep 指向的 PTE 置 0

`pte_modify(pte_t pte, pgprot_t newprot)`

用于修改页表项的低 12位控制域 (pgprot_t)，修改时其仅保留原控制域的 _PAGE_ACCESSED, _PAGE_MODIFIED 和 _CACHE_MASK 位，然后或上 newprot 即得新的 pte_t 的值。

```
pte_none(pte_t pte)
pte_present(pte_t pte)
pte_write(pte_t pte)
pte_dirty(pte_t pte)
pte_young(pte_t pte)
```

```
pte_file(pte_t pte)
```

其中：

```
static inline int pte_write(pte_t pte)  { return pte_val(pte) & _PAGE_WRITE; }
static inline int pte_dirty(pte_t pte)  { return pte_val(pte) & _PAGE_MODIFIED; }
static inline int pte_young(pte_t pte)  { return pte_val(pte) & _PAGE_ACCESSED; }
static inline int pte_file(pte_t pte)   { return pte_val(pte) & _PAGE_FILE; }
```

这些宏分别判断 pgprot 控制域是否置了 _PAGE_GLOBLE, _PAGE_PRESENT, _PAGE_WRITE, _PAGE_MODIFIED, _PAGE_ACCESSED, _PAGE_FILE，若置了相应位则返回 1，否则返回 0。

```
static inline pte_t pte_wrprotect(pte_t pte)
{
    pte_val(pte) &= ~(_PAGE_WRITE | _PAGE_SILENT_WRITE);
    return pte;
}

static inline pte_t pte_mkclean(pte_t pte)
{
    pte_val(pte) &= ~(_PAGE_MODIFIED|_PAGE_SILENT_WRITE);
    return pte;
}

static inline pte_t pte_mkold(pte_t pte)
{
    pte_val(pte) &= ~(_PAGE_ACCESSED|_PAGE_SILENT_READ);
    return pte;
}

static inline pte_t pte_mkwrite(pte_t pte)
{
    pte_val(pte) |= _PAGE_WRITE;
    if (pte_val(pte) & _PAGE_MODIFIED)
        pte_val(pte) |= _PAGE_SILENT_WRITE;
    return pte;
}
```

```
static inline pte_t pte_mkdirty(pte_t pte)
{
    pte_val(pte) |= _PAGE_MODIFIED;
    if (pte_val(pte) & _PAGE_WRITE)
        pte_val(pte) |= _PAGE_SILENT_WRITE;
    return pte;
}

static inline pte_t pte_mkyoung(pte_t pte)
{
    pte_val(pte) |= _PAGE_ACCESSED;
    if (pte_val(pte) & _PAGE_READ)
        pte_val(pte) |= _PAGE_SILENT_READ;
    return pte;
}

pte_wrprotect(pte_t pte)
pte_mkclean(pte_t pte)
pte_mkold(pte_t pte)
pte_mkwrite(pte_t pte)
pte_mkdirty(pte_t pte)
pte_mkyoung(pte_t pte)
```

这些函数分别用来置 pgprot 域的相应位。

```
static inline pgprot_t pgprot_noncached(pgprot_t _prot)
{
    unsigned long prot = pgprot_val(_prot);

    prot = (prot & ~_CACHE_MASK) | _CACHE_UNCACHED;

    return __pgprot(prot);
}
```

这个函数将 _prot 的 cache属性域置为 uncached，若某页表项的 cache属性域为 uncached 则该项对应的物理页在经 TLB 访问时，其页内数据是不会被缓存的

5.2 TLB 管理

TLB 管理的相关代码主要位于 [arch/mips/mm/tlbex.c] 和 [arch/mips/mm/tlb-r4k.c]。前者主要是 TLB 异常处理函数的实现，后者则主要是一系列 TLB 刷新函数的实现。

为了支持众多不同类型的处理器上不同的异常处理函数，MIPS Linux 在 mm 目录下实现了一个微型的汇编器，支持异常处理指令块的动态生成，这个实现位于 arch/mips/mm/uasm.c arch/mips/mm/uasm.h。基本思想很简单，关键的是细心加耐心，有兴趣的可以看看。

TLB 管理代码的初始化，位于 tlb_init()，调用拓补为：

```
trap_init()
     |--- ... ...
     |--- per_cpu_trap_init()
     `--- ...      |--- ... ...
                   |--- cpu_cache_init()
                   |--- tlb_init()
                   `--- ... ...
```

目前 MIPS 的 TLB 实现里主要有三种风格的实现：R3K，R4K 和 R8K。因此使用 arch/mips/mm/tlb-r3k.c, arch/mips/mm/tlb-r4k.c, arch/mips/mm/tlb-r8k.c 分别支持之，其中以 R4K 风格的实现使用最为广泛，我们就以 R4K 的为例，来看看其实现：

[arch/mips/mm/tlb-r4k.c]

```
void __cpuinit tlb_init(void)
{
    unsigned int config = read_c0_config();
```

```
    /*
     * You should never change this register:
     *   - On R4600 1.7 the tlbp never hits for pages smaller than
     *     the value in the c0_pagemask register.
     *   - The entire mm handling assumes the c0_pagemask register to
     *     be set to fixed-size pages.
     */
    probe_tlb(config);                      # 探测TLB的大小
    write_c0_pagemask(PM_DEFAULT_MASK);     # 根据页大小置PageMask寄存器
    write_c0_wired(0);                      # Wired 置 0
    write_c0_framemask(0);                  # FramMask 置 0
    temp_tlb_entry = current_cpu_data.tlbsize - 1;

        /* From this point on the ARC firmware is dead.  */
    local_flush_tlb_all();                  # 刷新整个本地 TLB

    /* Did I tell you that ARC SUCKS?  */

    if (ntlb) {                             # 处理无 TLB 的情形
        if (ntlb > 1 && ntlb <= current_cpu_data.tlbsize) {
            int wired = current_cpu_data.tlbsize - ntlb;
            write_c0_wired(wired);
            write_c0_index(wired-1);
            printk("Restricting TLB to %d entries\n", ntlb);
        } else
            printk("Ignoring invalid argument ntlb=%d\n", ntlb);
    }

    build_tlb_refill_handler();           # 生成所有 TLB 异常处理函数
}
```

这个 build_tlb_refill_handler() 定义于 [arch/mips/mm/tlbex.c]:

```
oid __cpuinit build_tlb_refill_handler(void)
{
    /*
     * The refill handler is generated per-CPU, multi-node systems
     * may have local storage for it. The other handlers are only
```

```
     * needed once.
     */
    static int run_once = 0;

    switch (current_cpu_type()) {
    case CPU_R2000:
    case CPU_R3000:
    case CPU_R3000A:
    case CPU_R3081E:
    case CPU_TX3912:
    case CPU_TX3922:
    case CPU_TX3927:
        build_r3000_tlb_refill_handler();
        if (!run_once) {
            build_r3000_tlb_load_handler();
            build_r3000_tlb_store_handler();
            build_r3000_tlb_modify_handler();
            run_once++;
        }
        break;

    case CPU_R6000:
    case CPU_R6000A:
        panic("No R6000 TLB refill handler yet");
        break;

    case CPU_R8000:
        panic("No R8000 TLB refill handler yet");
        break;

    default:
        build_r4000_tlb_refill_handler();
        if (!run_once) {
            build_r4000_tlb_load_handler();
            build_r4000_tlb_store_handler();
            build_r4000_tlb_modify_handler();
            run_once++;
        }
    }
```

```
}
```

其中 build_r4000_tlb_refill_handler() 负责生成 TLB Refill 异常处理函数，入口是专用的 TLB Refill 异常入口 (0x80000000)。
build_r4000_tlb_load_handler() 负责生成 TLB Load 异常的处理函数，这个异常是 TLB Invalid 异常且ExeCode 为 2 (TLB Load)，入口是通用异常的入口。
build_r4000_tlb_store_handler() 负责生成 TLB Store 异常的处理函数，这个异常是 TLB Invalid 异常，ExeCode 为 3 (TLB Store)，入口是通用异常的入口。
build_r4000_tlb_modify_handler() 则负责生成 TLB Modified 异常的处理函数，这个异常就是 TLB Modified 异常，ExeCode 为 1，入口是通用异常的入口。

5.2.1 TLB Refill

TLB Refiil handler 由函数 build_r4000_tlb_refill_handler() 负责生成，并将其拷贝到异常入口处，该函数定义于 [arch/mips/mm/tlbex.c]:

```
656  static void __cpuinit build_r4000_tlb_refill_handler(void)
657  {
658      u32 *p = tlb_handler;
659      struct uasm_label *l = labels;
660      struct uasm_reloc *r = relocs;
661      u32 *f;
662      unsigned int final_len;
663
664      memset(tlb_handler, 0, sizeof(tlb_handler));
665      memset(labels, 0, sizeof(labels));
666      memset(relocs, 0, sizeof(relocs));
667      memset(final_handler, 0, sizeof(final_handler));
668
669      /*
670       * create the plain linear handler
671       */
672      if (bcm1250_m3_war()) {
673          UASM_i_MFC0(&p, K0, C0_BADVADDR);
674          UASM_i_MFC0(&p, K1, C0_ENTRYHI);
675          uasm_i_xor(&p, K0, K0, K1);
676          UASM_i_SRL(&p, K0, K0, PAGE_SHIFT + 1);
677          uasm_il_bnez(&p, &r, K0, label_leave);
678          /* No need for uasm_i_nop */
679      }
680
681  #ifdef CONFIG_64BIT
682      build_get_pmde64(&p, &l, &r, K0, K1); /* get pmd in K1 */
683  #else
```

```
684      build_get_pgde32(&p, K0, K1); /* get pgd in K1 */
685  #endif
686
```

以 32 位的情形为例，概要概述一下。
tlb_handler 定义为：

```
147  static u32 tlb_handler[128] __cpuinitdata;
```

是为存放 TLB 异常处理函数的一个缓存，大小为 128 * 4 字节，可以存放 128 条指令。
669 ~ 679 处理 BCM1250 的特殊实现，这里我们不关心。
684 行的 build_get_pgde32(&p, K0, K1) 则负责生成一段指令，并将其存放于 tlb_handler，该段指令是为获取转换失败的地址所对应的 PGD 表项入口，寄存器 K1 存放所读取的 PGD 表项入口地址：

```
527  static void __cpuinit __maybe_unused
528  build_get_pgde32(u32 **p, unsigned int tmp, unsigned int ptr)
529  {
530      long pgdc = (long)pgd_current;
531
532      /* 32 bit SMP has smp_processor_id() stored in CONTEXT. */
533  #ifdef CONFIG_SMP
534  #ifdef  CONFIG_MIPS_MT_SMTC
535      /*
536       * SMTC uses TCBind value as "CPU" index
537       */
538      uasm_i_mfc0(p, ptr, C0_TCBIND);
539      UASM_i_LA_mostly(p, tmp, pgdc);
540      uasm_i_srl(p, ptr, ptr, 19);
541  #else
542      /*
543       * smp_processor_id() << 3 is stored in CONTEXT.
544       */
545      uasm_i_mfc0(p, ptr, C0_CONTEXT);
546      UASM_i_LA_mostly(p, tmp, pgdc);
547      uasm_i_srl(p, ptr, ptr, 23);
548  #endif
549      uasm_i_addu(p, ptr, tmp, ptr);
550  #else
551      UASM_i_LA_mostly(p, ptr, pgdc);
552  #endif
553      uasm_i_mfc0(p, tmp, C0_BADVADDR); /* get faulting address */
554      uasm_i_lw(p, ptr, uasm_rel_lo(pgdc), ptr);
555      uasm_i_srl(p, tmp, tmp, PGDIR_SHIFT); /* get pgd only bits */
556      uasm_i_sll(p, tmp, tmp, PGD_T_LOG2);
557      uasm_i_addu(p, ptr, ptr, tmp); /* add in pgd offset */
558  }
```

pgd_current 定义于 arch/mips/mm/init.c:

```
unsigned long pgd_current[NR_CPUS];
```

用于存放每个 CPU 当前进程对应的 PGD 入口地址。545 ~ 547 则是在 SMP 时，从 Context 寄存器中取 smp_processor_id()。Linux 设计时是将 smp_processor_id() 左移 2 位存入 Context 寄存器的 PTEBase 域，因此每次取 Context 的值右移 23 位即得 smp_processor_id() << 2 的值。留意这些定义：

[include/asm-mips/mmu_context.h]:

```
#define TLBMISS_HANDLER_SETUP_PGD(pgd) \
    pgd_current[smp_processor_id()] = (unsigned long)(pgd)

#ifdef CONFIG_32BIT
#define TLBMISS_HANDLER_SETUP()                     \
    write_c0_context((unsigned long) smp_processor_id() << 25); \
    TLBMISS_HANDLER_SETUP_PGD(swapper_pg_dir)
#endif
#ifdef CONFIG_64BIT
#define TLBMISS_HANDLER_SETUP()                     \
    write_c0_context((unsigned long) smp_processor_id() << 26); \
    TLBMISS_HANDLER_SETUP_PGD(swapper_pg_dir)
#endif
```

宏 TLBMISS_HANDLER_SETUP() 在每个 CPU 初始化时调用 per_cpu_trap_init() 时调用。可以看到 32 位时 Context[PTEBase] 存放的是 smp_processor_id() << 2，64 位时 Context[PTEBase] 存放的是 smp_processor_id() << 3。
宏 TLBMISS_HANDLER_SETUP_PDG(pgd) 则在每次上下文切换是调用，用于将进程的 PGD 入口地址置入 pgd_current[smp_processor_id()]。

549 行等价于 &pgd_current[smp_processor_id()]，即是计算指向 pgd_current[smp_processor_id()] 的指针。

554 行则是取 pgd_current[smp_processor_id()] 的内容，即当前进程的 PGD 入口地

址。

553, 555, 556 行则是从 BadVAddr 中获取转换失败的虚拟地址，移位获取索引 PGD 的偏移。

最后 557 行将当前进程 PGD 入口地址与索引 PGD 的偏移相加，即得转换失败的地址所对应的 PGD 表项入口地址，这个地址放在 K1 中。

接着 build_r4000_tlb_refill_handler():

```
687       build_get_ptep(&p, K0, K1);
688       build_update_entries(&p, K0, K1);
689       build_tlb_write_entry(&p, &l, &r, tlb_random);
```

build_get_ptep(&p, K0, K1) 负责生成一段指令，并将其存放于 tlb_handler，该段指令是为获取转换失败的地址所对应的页表项入口，寄存器 K1 存放所读取的页表项入口地址：

```
588  static void build_get_ptep(u32 **p, unsigned int tmp, unsigned int ptr)
589  {
590       /*
591        * Bug workaround for the Nevada. It seems as if under certain
592        * circumstances the move from cp0_context might produce a
593        * bogus result when the mfc0 instruction and its consumer are
594        * in a different cacheline or a load instruction, probably any
595        * memory reference, is between them.
596        */
597       switch (current_cpu_type()) {
598       case CPU_NEVADA:
599            UASM_i_LW(p, ptr, 0, ptr);
600            GET_CONTEXT(p, tmp);
601            break;
602
603       default:
604            GET_CONTEXT(p, tmp);          # 取 Context 的值入寄存器 K0
605            UASM_i_LW(p, ptr, 0, ptr);    # 取指针 K1 处的一个字，入 K1
606            break;
607       }
608
609       build_adjust_context(p, tmp);
610       UASM_i_ADDU(p, ptr, ptr, tmp);
611  }
```

605 行实际取得是上一步 K1 所指向的 PGD 项。build_adjust_context(p, tmp) 生成一段指令，负责将 K0 中的 Context 值移位得转换失败的虚址对应的页表内偏移，这个偏移

存放于 K0 中。

610 行 K1 + K0 即得转换失败的地址所对应的页表项入口地址，其置 K1 中。

```
133  #ifdef CONFIG_64BIT
134  # define GET_CONTEXT(buf, reg) UASM_i_MFC0(buf, reg, C0_XCONTEXT)
135  #else
136  # define GET_CONTEXT(buf, reg) UASM_i_MFC0(buf, reg, C0_CONTEXT)
137  #endif
```

```
562  static void build_adjust_context(u32 **p, unsigned int ctx)
563  {
564      unsigned int shift = 4 - (PTE_T_LOG2 + 1) + PAGE_SHIFT - 12;
565      unsigned int mask = (PTRS_PER_PTE / 2 - 1) << (PTE_T_LOG2 + 1);
566
567      switch (current_cpu_type()) {
568      case CPU_VR41XX:
569      case CPU_VR4111:
570      case CPU_VR4121:
571      case CPU_VR4122:
572      case CPU_VR4131:
573      case CPU_VR4181:
574      case CPU_VR4181A:
575      case CPU_VR4133:
576          shift += 2;
577          break;
578
579      default:
580          break;
581      }
582
583      if (shift)
584          UASM_i_SRL(p, ctx, ctx, shift);
585      uasm_i_andi(p, ctx, ctx, mask);
586  }
```

build_update_entries() 则生成一段指令，负责将从对应的页表项入口读取两个页表项（相邻的奇偶页），右移 6 位后写入 EntryLo0 和 EntryLo1：

```
613  static void build_update_entries(u32 **p, unsigned int tmp,
614                       unsigned int ptep)
615  {
616      /*
617       * 64bit address support (36bit on a 32bit CPU) in a 32bit
618       * Kernel is a special case. Only a few CPUs use it.
619       */
620  #ifdef CONFIG_64BIT_PHYS_ADDR
621      if (cpu_has_64bits) {
622          uasm_i_ld(p, tmp, 0, ptep); /* get even pte */
623          uasm_i_ld(p, ptep, sizeof(pte_t), ptep); /* get odd pte */
624          uasm_i_dsrl(p, tmp, tmp, 6); /* convert to entrylo0 */
625          uasm_i_mtc0(p, tmp, C0_ENTRYLO0); /* load it */
```

```
626             uasm_i_dsrl(p, ptep, ptep, 6); /* convert to entrylo1 */
627             uasm_i_mtc0(p, ptep, C0_ENTRYLO1); /* load it */
628         } else {
629             int pte_off_even = sizeof(pte_t) / 2;
630             int pte_off_odd = pte_off_even + sizeof(pte_t);
631
632             /* The pte entries are pre-shifted */
633             uasm_i_lw(p, tmp, pte_off_even, ptep); /* get even pte */
634             uasm_i_mtc0(p, tmp, C0_ENTRYLO0); /* load it */
635             uasm_i_lw(p, ptep, pte_off_odd, ptep); /* get odd pte */
636             uasm_i_mtc0(p, ptep, C0_ENTRYLO1); /* load it */
637         }
638     #else
639         UASM_i_LW(p, tmp, 0, ptep);              /* get even pte */
640         UASM_i_LW(p, ptep, sizeof(pte_t), ptep); /* get odd pte */
641         if (r45k_bvahwbug())
642             build_tlb_probe_entry(p);
643         UASM_i_SRL(p, tmp, tmp, 6);          /* convert to entrylo0 */
644         if (r4k_250MHZhwbug())
645             uasm_i_mtc0(p, 0, C0_ENTRYLO0);
646         uasm_i_mtc0(p, tmp, C0_ENTRYLO0);    /* load it */
647         UASM_i_SRL(p, ptep, ptep, 6);        /* convert to entrylo1 */
648         if (r45k_bvahwbug())
649             uasm_i_mfc0(p, tmp, C0_INDEX);
650         if (r4k_250MHZhwbug())
651             uasm_i_mtc0(p, 0, C0_ENTRYLO1);
652         uasm_i_mtc0(p, ptep, C0_ENTRYLO1);   /* load it */
653     #endif
654     }
```

至此写 TLB 项前，EntryLo0 和 EntryLo1 已经准备好了，往下就是直接调用 build_tlb_write_entry(&p, &l, &r, tlb_random) 以随机的方式写 TLB (tlbwr) 。往后的就是一些后续的处理了，包括 eret 异常返回了，最后 build 过程，还要检查一下生成的 TLB Refill handler 是否超出限定的长度（体系结构的规定）：

```
690         uasm_l_leave(&l, p);
691         uasm_i_eret(&p); /* return from trap */
692
693     #ifdef CONFIG_64BIT
694         build_get_pgd_vmalloc64(&p, &l, &r, K0, K1);
695     #endif
696
697         /*
698          * Overflow check: For the 64bit handler, we need at least one
699          * free instruction slot for the wrap-around branch. In worst
700          * case, if the intended insertion point is a delay slot, we
701          * need three, with the second nop'ed and the third being
702          * unused.
703          */
704         /* Loongson2 ebase is different than r4k, we have more space */
705     #if defined(CONFIG_32BIT) || defined(CONFIG_CPU_LOONGSON2)
706         if ((p - tlb_handler) > 64)
707             panic("TLB refill handler space exceeded");
```

```
708 #else
709     if (((p - tlb_handler) > 63)
710         || (((p - tlb_handler) > 61)
711         && uasm_insn_has_bdelay(relocs, tlb_handler + 29)))
712         panic("TLB refill handler space exceeded");
713 #endif
714
715     /*
716      * Now fold the handler in the TLB refill handler space.
717      */
718 #if defined(CONFIG_32BIT) || defined(CONFIG_CPU_LOONGSON2)
719     f = final_handler;
720     /* Simplest case, just copy the handler. */
721     uasm_copy_handler(relocs, labels, tlb_handler, p, f);
722     final_len = p - tlb_handler;
723 #else /* CONFIG_64BIT */
724     f = final_handler + 32;
725     if ((p - tlb_handler) <= 32) {
726         /* Just copy the handler. */
727         uasm_copy_handler(relocs, labels, tlb_handler, p, f);
728         final_len = p - tlb_handler;
729     } else {
730         u32 *split = tlb_handler + 30;
731
732         /*
733          * Find the split point.
734          */
735         if (uasm_insn_has_bdelay(relocs, split - 1))
736             split--;
737
738         /* Copy first part of the handler. */
739         uasm_copy_handler(relocs, labels, tlb_handler, split, f);
740         f += split - tlb_handler;
741
742         /* Insert branch. */
743         uasm_l_split(&l, final_handler);
744         uasm_il_b(&f, &r, label_split);
745         if (uasm_insn_has_bdelay(relocs, split))
746             uasm_i_nop(&f);
747         else {
748             uasm_copy_handler(relocs, labels, split, split + 1, f);
749             uasm_move_labels(labels, f, f + 1, -1);
750             f++;
751             split++;
752         }
753
754         /* Copy the rest of the handler. */
755         uasm_copy_handler(relocs, labels, split, p, final_handler);
756         final_len = (f - (final_handler + 32)) + (p - split);
757     }
758 #endif /* CONFIG_64BIT */
759
760     uasm_resolve_relocs(relocs, labels);
761     pr_debug("Wrote TLB refill handler (%u instructions).\n",
762          final_len);
763
764     memcpy((void *)ebase, final_handler, 0x100);
765
```

```
766       dump_handler((u32 *)ebase, 64);
767   }
```

最后则是将已经生成的异常处理指令直接 memcpy 到 ebase 处，这个 ebase 就是 TLB Refill 异常的入口，32 位下其值为 0x8000 0000。

5.2.2 TLB Flush

[arch/mips/mm/tlb-r4k.c]

```
67  void local_flush_tlb_all(void)
68  {
69      unsigned long flags;
70      unsigned long old_ctx;
71      int entry;
72
73      ENTER_CRITICAL(flags);
74      /* Save old context and create impossible VPN2 value */
75      old_ctx = read_c0_entryhi();
76      write_c0_entrylo0(0);
77      write_c0_entrylo1(0);
78
79      entry = read_c0_wired();
80
81      /* Blast 'em all away. */
82      while (entry < current_cpu_data.tlbsize) {
83          /* Make sure all entries differ. */
84          write_c0_entryhi(UNIQUE_ENTRYHI(entry));
85          write_c0_index(entry);
86          mtc0_tlbw_hazard();
87          tlb_write_indexed();
88          entry++;
89      }
90      tlbw_use_hazard();
91      write_c0_entryhi(old_ctx);
92      FLUSH_ITLB;
93      EXIT_CRITICAL(flags);
94  }
```

所谓 Flush TLB 就是将 TLB 的所有非固定 (wired) 项置 0

5.3 ASID 管理

MIPS 用 8 位 ASID 来区分 TLB 表项所属的进程，但当进程数超过 256 时，这个位数是不够用的，Linux 的设计是用软件扩展。

ASID 的管理代码位于 include/asm-mips/mmu_context.h，特别留意内核对 get_new_mmu_context() 的上下文环境。

每个 mm_struct 里有一个 context[cpu] 成员，用于存储软件扩展的 asid_cache ，当前所有 cpu 的 asid_cache 存储在 cpu_data[cpu].asid_cache 里。

switch_mm 时，其皆会检查新的 mm_struct 之 context[cpu] 与当前 asid_cache 的 version 是否相等，如相等，则说明当前 TLB 中的所有项的所有者与新成员同属一个组，则无需 flush TLB；若不等，则其不属同一组，则可能存在低 8 位重名，则调用 get_new_mmu_context() 获取一个新的 asid_cache （以当前 cpu_data[cpu].asid_cache 为基础加 1），若新 asid_cache 与当前 asid_cache 不在同一组，则还需要 flush TLB（即以新 asid_cache 为当前组），尔后 mm_struct->context[cpu] = cpu_data[cpu].asid_cache = asid_cache & 0xff ；新 asid_cache 与当前 asid_cache 在同一组，因新 asid_cache 并无其它进程使用（同一组的 asid_cache ，其低 8 位都是顺序获取的），故无需 flush TLB 。

在 fork 一个新进程时，皆直接调用 get_new_mmu_context() 获取一个新的 asid_cache 。

mm->cpu_vm_mask 是一个 bitmap，相应位置 1，表示进程在 CPU x 上运行。如 cpu_vm_mask 之 15 位置 1，则说明进程在 CPU 15 上运行。