蜗窝科技

慢下来，享受技术。

进程切换分析（2）：TLB处理

作者：linuxer 发布于：2017-2-9 12:05 分类：进程管理

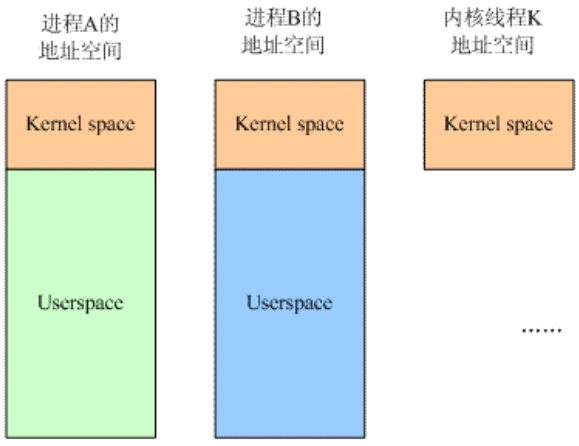
一、前言

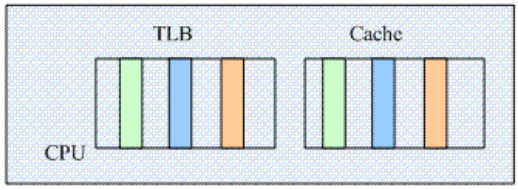
进程切换是一个复杂的过程，本文不准备详细描述整个进程切换的方方面面，而是关注进程切换中一个小小的知识点：TLB的处理。为了能够讲清楚这个问题，我们在第二章描述在单CPU场景下一些和TLB相关的细节，第三章推进到多核场景，至此，理论部分结束。在第二章和第三章，我们从基本的逻辑角度出发，并不拘泥于特定的CPU和特定的OS，这里需要大家对基本的TLB的组织原理有所了解，具体可以参考本站的《TLB操作》一文。再好的逻辑也需要体现在HW block和SW block的设计中，在第四章，我们给出了linux4.4.6内核在ARM64平台上的TLB代码处理细节（在描述tlb lazy mode的时候引入部分x86架构的代码），希望能通过具体的代码和实际的CPU硬件行为加深大家对原理的理解。

二、单核场景的工作原理

1、block diagram

我们先看看在单核场景下，和进程切换相关的逻辑block示意图：





CPU上运行了若干的用户空间的进程和内核线程，为了加快性能，CPU中往往设计了TLB和Cache这样的HW block。Cache为了更快的访问main memory中的数据和指令，而TLB是为了更快的进行地址翻译而将部分的页表内容缓存到了Translation lookasid buffer中，避免了从main memory访问页表的过程。

站内搜索
请输入关键词 搜索

功能
留言板
评论列表
支持者列表

最新评论
LLEo
感谢wowo 大佬
yz
@无非：group0和group1其中一个可以产生fiq，如...
xdwinter
聊表心意~感谢蜗窝,收益颇多。
xdwinter
聊表心意~感谢蜗窝
little_vage
向蜗蜗大佬致敬。不忘初心，牢记使命!
ttdevrs
图片很棒

文章分类
Linux内核分析(23)
统一设备模型(15)
电源管理子系统(43)
中断子系统(15)
进程管理(29)
内核同步机制(22)
GPIO子系统(5)
时间子系统(14)
通信类协议(7)
内存管理(31)
图形子系统(2)
文件系统(5)
TTY子系统(6)
u-boot分析(4)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
蓝牙(16)
ARMv8A Arch(15)
显示(3)
USB(1)
基础学科(10)
技术漫谈(12)

假如不做任何的处理，那么在进程A切换到进程B的时候，TLB和Cache中同时存在了A和B进程的数据。对于kernel space其实无所谓，因为所有的进程都是共享的，但是对于A和B进程，它们各种有自己的独立的用户地址空间，也就是说，同样的一个虚拟地址X，在A的地址空间中可以被翻译成Pa，而在B地址空间中会被翻译成Pb，如果在地址翻译过程中，TLB中同时存在A和B进程的数据，那么旧的A地址空间的缓存项会影响B进程地址空间的翻译，因此，在进程切换的时候，需要有tlb的操作，以便清除旧进程的影响，具体怎样做呢？我们下面一一讨论。

2、绝对没有问题，但是性能不佳的方案

当系统发生进程切换，从进程A切换到进程B，从而导致地址空间也从A切换到B，这时候，我们可以认为在A进程执行过程中，所有TLB和Cache的数据都是for A进程的，一旦切换到B，整个地址空间都不一样了，因此需要全部flush掉（注意：我这里使用了linux内核的术语，flush就是意味着将TLB或者cache中的条目设置为无效，对于一个ARM平台上的嵌入式工程师，一般我们会更习惯使用invalidate这个术语，不管怎样，在本文中，flush等于invalidate）。

这种方案当然没有问题，当进程B被切入执行的时候，其面对的CPU是一个干干净净，从头开始的硬件环境，TLB和Cache中不会有任何的残留的A进程的数据来影响当前B进程的执行。当然，稍微有一点遗憾的就是在B进程开始执行的时候，TLB和Cache都是冰冷的（空空如也），因此，B进程刚开始执行的时候，TLB miss和Cache miss都非常严重，从而导致了性能的下降。

3、如何提高TLB的性能?

对一个模块的优化往往需要对该模块的特性进行更细致的分析、归类，上一节，我们采用进程地址空间这样的术语，其实它可以被进一步细分为内核地址空间和用户地址空间。对于所有的进程（包括内核线程），内核地址空间是一样的，因此对于这部分地址翻译，无论进程如何切换，内核地址空间转换到物理地址的关系是永远不变的，其实在进程A切换到B的时候，不需要flush掉，因为B进程也可以继续使用这部分的TLB内容（上图中，橘色的block）。对于用户地址空间，各个进程都有自己独立的地址空间，在进程A切换到B的时候，TLB中的和A进程相关的entry（上图中，青色的block）对于B是完全没有任何意义的，需要flush掉。

在这样的思路指导下，我们其实需要区分global和local（其实就是process-specific的意思）这两种类型的地址翻译，因此，在页表描述符中往往有一个bit来标识该地址翻译是global还是local的，同样的，在TLB中，这个标识global还是local的flag也会被缓存起来。有了这样的设计之后，我们可以根据不同的场景而flush all或者只是flush local tlb entry。

4、特殊情况的考量

我们考虑下面的场景：进程A切换到内核线程K之后，其实地址空间根本没有必要切换，线程K能访问的就是内核空间的那些地址，而这些地址也是和进程A共享的。既然没有切换地址空间，那么也就不需要flush 那些进程特定的tlb entry了，当从K切换会A进程后，那么所有TLB的数据都是有效的，从大大降低了tlb miss。此外，对于多线程环境，切换可能发生在一个进程中的两个线程，这时候，线程在同样的地址空间，也根本不需要flush tlb。

4、进一步提升TLB的性能

还有可能进一步提升TLB的性能吗？有没有可能根本不flush TLB?

当然可以，不过这需要我们在设计TLB block的时候需要识别process specific的tlb entry，也就是说，TLB block需要感知到各个进程的地址空间。为了完成这样的设计，我们需要标识不同的address space，这里有一个术语叫做ASID（address space ID）。原来TLB查找是通过虚拟地址VA来判断是否TLB hit。有了ASID的支持后，TLB hit的判断标准修改为（虚拟地址＋ASID），ASID是每一个进程分配一个，标识自己的进程地址空间。TLB block如何知道一个tlb entry的ASID呢？一般会来自CPU的系统寄存器（对于ARM64平台，它来自TTBRx_EL1寄存器），这样在TLB block在缓存（VA-PA-Global flag）的同时，也就把当前的ASID缓存在了对应的TLB entry中，这样一个TLB entry中包括了（VA-PA-Global flag-ASID）。

有了ASID的支持后，A进程切换到B进程再也不需要flush tlb了，因为A进程执行时候缓存在TLB中的残留A地址空间相关的entry不会影响到B进程，虽然A和B可能有相同的VA，但是ASID保证了硬件可以区分A和B进程地址空间。

三、多核的TLB操作

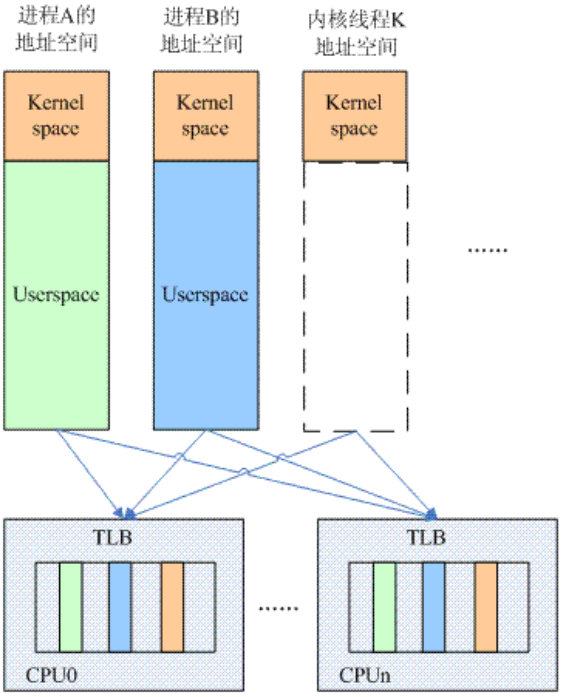
1、block diagram

完成单核场景下的分析之后，我们一起来看看多核的情况。进程切换相关的TLB逻辑block示意图如下：

项目专区(0)
X Project(28)

随机文章
蜗窝流量地域统计
内存初始化代码分析（二）：内存布局
Linux common clock framework(2)_clock provider
页面回收的基本概念
Linux内核同步机制之（六）：Seqlock

文章存档
2022年2月(2)
2022年1月(1)
2021年12月(1)
2021年11月(5)
2021年7月(1)
2021年6月(1)
2021年5月(3)
2020年3月(3)
2020年2月(2)
2020年1月(3)
2019年12月(3)
2019年5月(4)
2019年3月(1)
2019年1月(3)
2018年12月(2)
2018年11月(1)
2018年10月(2)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(6)



在多核系统中，进程切换的时候，TLB的操作要复杂一些，主要原因有两点：其一是各个cpu core有各自的TLB，因此TLB的操作可以分成两类，一类是flush all，即将所有cpu core上的tlb flush掉，还有一类操作是flush local tlb，即仅仅flush本cpu core的tlb。另外一个原因是进程可以调度到任何一个cpu core上执行（当然具体和cpu affinity的设定相关），从而导致task处处留情（在各个cpu上留有残余的tlb entry）。

2、TLB操作的基本思考

根据上一节的描述，我们了解到地址翻译有global（各个进程共享）和local（进程特定的）的概念，因而tlb entry也有global和local的区分。如果不区分这两个概念，那么进程切换的时候，直接flush该cpu上的所有残余。这样，当进程A切出的时候，留给下一个进程B一个清爽的tlb，而当进程A在其他cpu上再次调度的时候，它面临的也是一个全空的TLB（其他cpu的tlb不会影响）。当然，如果区分global 和local，那么tlb操作也基本类似，只不过进程切换的时候，不是flush该cpu上的所有tlb entry，而是flush所有的tlb local entry就OK了。

对local tlb entry还可以进一步细分，那就是了ASID（address space ID）或者PCID（process context ID）的概念了（global tlb entry不区分ASID）。如果支持ASID（或者PCID）的话，tlb操作变得简单一些，或者说我们没有必要执行tlb操作了，因为在TLB搜索的时候已经可以区分各个task上下文了，这样，各个cpu中残留的tlb不会影响其他任务的执行。在单核系统中，这样的操作可以获取很好的性能。比如A---B--->A这样的场景中，如果TLB足够大，可以容纳2个task的tlb entry（现代cpu一般也可以做到这一点），那么A再次切回的时候，TLB是hot的，大大提升了性能。

不过，对于多核系统，这种情况有一点点的麻烦，其实也就是传说中的TLB shootdown带来的性能问题。在多核系统中，如果cpu支持PCID并且在进程切换的时候不flush tlb，那么系统中各个cpu中的tlb entry则保留各种task的tlb entry，当在某个cpu上，一个进程被销毁，或者修改了自己的页表（也就是修改了VA PA映射关系）的时候，我们必须将该task的相关tlb entry从系统中清除出去。这时候，你不仅仅需要flush本cpu上对应的TLB entry，还需要shootdown其他cpu上的和该task相关的tlb残余。而这个动作一般是通过IPI实现（例如X86），从而引入了开销。此外PCID的分配和管理也会带来额外的开销，因此，OS是否支持PCID（或者ASID）是由各个arch代码自己决定（对于linux而言，x86不支持，而ARM平台是支持的）。

四、进程切换中的tlb操作代码分析

1、tlb lazy mode

在context_switch中有这样的一段代码：

```
if (!mm) {
    next->active_mm = oldmm;
    atomic_inc(&oldmm->mm_count);
    enter_lazy_tlb(oldmm, next);
} else
    switch_mm(oldmm, mm, next);
```

2015年4月(9)
2015年3月(9)
2015年2月(6)
2015年1月(6)
2014年12月(17)
2014年11月(8)
2014年10月(9)
2014年9月(7)
2014年8月(12)
2014年7月(6)
2014年6月(6)
2014年5月(9)
2014年4月(9)
2014年3月(7)
2014年2月(3)
2014年1月(4)

这段代码的意思就是如果要切入的next task是一个内核线程（next->mm == NULL ）的话，那么可以通过enter_lazy_tlb函数标记本cpu上的next task进入lazy TLB mode。由于ARM64平台上的enter_lazy_tlb函数是空函数，因此我们采用X86来描述lazy TLB mode。

当然，我们需要一些准备工作，毕竟对于熟悉ARM平台的嵌入式工程师而言，x86多少有点陌生。

到目前，我们还都是从逻辑角度来描述TLB操作，但是在实际中，进程切换中的tlb操作是HW完成还是SW完成呢？不同的处理器思路是不一样的（具体原因未知），有的处理器是HW完成，例如X86，在加载cr3寄存器进行地址空间切换的时候，hw会自动操作tlb。而有的处理是需要软件参与完成tlb操作，例如ARM系列的处理器，在切换TTBR寄存器的时候，HW没有tlb动作，需要SW完成tlb操作。因此，x86平台上，在进程切换的时候，软件不需要显示的调用tlb flush函数，在switch_mm函数中会用next task中的mm->pgd加载CR3寄存器，这时候load cr3的动作会导致本cpu中的local tlb entry被全部flush掉。

在x86支持PCID（X86术语，相当与ARM的ASID）的情况下会怎样呢？也会在load cr3的时候flush掉所有的本地CPU上的 local tlb entry吗？其实在linux中，由于TLB shootdown，普通的linux并不支持PCID（KVM中会使用，但是不在本文考虑范围内），因此，对于x86的进程地址空间切换，它就是会有flush local tlb entry这样的side effect。

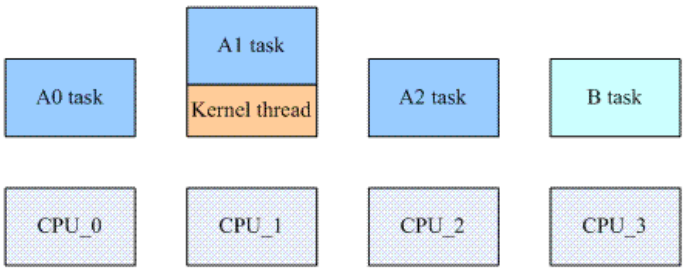
另外有一点是ARM64和x86不同的地方：ARM64支持在一个cpu core执行tlb flush的指令，例如tlbi vmalle1is，将inner shareablity domain中的所有cpu core的tlb全部flush掉。而x86不能，如果想要flush掉系统中多有cpu core的tlb，只能是通过IPI通知到其他cpu进行处理。

好的，至此，所有预备知识都已经ready了，我们进入tlb lazy mode这个主题。虽然进程切换伴随tlb flush操作，但是某些场景亦可避免。在下面的场景，我们可以不flush tlb（我们仍然采用A--->B task的场景来描述）：

（1）如果要切入的next task B是内核线程，那么我们也暂时不需要flush TLB，因为内核线程不会访问usersapce，而那些进程A残留的TLB entry也不会影响内核线程的执行，毕竟B没有自己的用户地址空间，而且和A共享内核地址空间。

（2）如果A和B在一个地址空间中（一个进程中的两个线程），那么我们也暂时不需要flush TLB。

除了进程切换，还有其他的TLB flush场景。我们先看一个通用的TLB flush场景，如下图所示：



一个4核系统中，A0 A1和A2 task属于同一个进程地址空间，CPU_0和CPU_2上分别运行了A0和A2 task，CPU_1有点特殊，它正在运行一个内核线程，但是该内核线程正在借用A1 task的地址空间，CPU_3上运行不相关的B task。

当A0 task修改了自己的地址翻译，那么它不能只是flush CPU_0的tlb，还需要通知到CPU_1和CPU_2，因为这两个CPU上当前active的地址空间和CPU_0是一样的。由于A1 task的修改，CPU_1和CPU_2上的这些缓存的TLB entry已经失效了，需要flush。同理，可以推广到更多的CPU上，也就是说，在某个CPUx上运行的task修改了地址映射关系，那么tlb flush需要传递到所有相关的CPU中（当前的mm等于CPUx的current mm）。在多核系统中，这样的通过IPI来传递TLB flush的消息会随着cpu core的增加而增加，有没有办法减少那些没有必要的TLB flush呢？当然有，也就是上图中的A1 task场景，这也就是传说中的lazy tlb mode。

我先回头看看代码。在代码中，如果next task是内核线程，我们并不会执行switch_mm（该函数会引起tlb flush的动作），而是调用enter_lazy_tlb进入lazy tlb mode。在x86架构下，代码如下：

```
static inline void enter_lazy_tlb(struct mm_struct *mm, struct task_struct *tsk)
{
#ifdef CONFIG_SMP
    if (this_cpu_read(cpu_tlbstate.state) == TLBSTATE_OK)
        this_cpu_write(cpu_tlbstate.state, TLBSTATE_LAZY);
#endif
}
```

在x86架构下，进入lazy tlb mode也就是在该cpu的cpu_tlbstate变量中设定TLBSTATE_LAZY的状态就OK了。因此，进入lazy mode的时候，也就不需要调用switch_mm来切换进程地址空间，也就不会执行flush tlb这样毫无意义的动作了。enter_lazy_tlb并不操作硬件，只要记录该cpu的软件状态就OK了。

切换之后，内核线程进入执行状态，CPU_1的TLB残留进程A的entry，这对于内核线程的执行没有影响，但是当其他CPU发送IPI要求flush TLB的时候呢？按理说应该立刻flush tlb，但是在lazy tlb mode下，我们可以不执行flush tlb操作。这样问题来了：什么时候flush掉残留的A进程的tlb entry呢？答案是在下一次进程切换中。因为一旦内核线程被schedule out，并且切入一个新的进程C，那么在switch_mm，切入到C进程地址空间的时候，所有之前的残留都会被清除掉（因为有load cr3的动作）。因此，在执行内核线程的时候，我们可以推迟tlb invalidate的请求。也就是说，当收到ipi中断要求进行该mm的tlb invalidate的动作的时候，我们暂时没有必要执行了，只需要记录状态就OK了。

2、ARM64中如何管理ASID？

和x86不同的是：ARM64支持了ASID（类似x86的PCID），难道ARM64解决了TLB Shootdown的问题？其实我也在思考这个问题，但是还没有想明白。很显然，在ARM64中，我们不需要通过IPI来进行所有cpu core的TLB flush动作，ARM64在指令集层面支持shareable domain中所有PEs上的TLB flush动作，也许是这样的指令让TLB flush的开销也没有那么大，那么就可以选择支持ASID，在进程切换的时候不需要进行任何的TLB操作，同时，由于不需要IPI来传递TLB flush，那么也就没有特别的处理lazy tlb mode了。

既然linux中，ARM64选择支持ASID，那么它就要直面ASID的分配和管理问题了。硬件支持的ASID有一定限制，它的编址空间是8个或者16个bit，最大256或者65535个ID。当ASID溢出之后如何处理呢？这就需要一些软件的控制来协调处理。 我们用硬件支持上限为256个ASID的情景来描述这个基本的思路：当系统中各个cpu的TLB中的asid合起来不大于256个的时候，系统正常运行，一旦超过256的上限后，我们将全部TLB flush掉，并重新分配ASID，每达到256上限，都需要flush tlb并重新分配HW ASID。具体分配ASID代码如下：

```
static u64 new_context(struct mm_struct *mm, unsigned int cpu)
{
    static u32 cur_idx = 1;
    u64 asid = atomic64_read(&mm->context.id);
    u64 generation = atomic64_read(&asid_generation);

    if (asid != 0) { - - - - - - - - - - - - - - - - - - - - - - - - （1）
        u64 newasid = generation | (asid & ~ASID_MASK);
        if (check_update_reserved_asid(asid, newasid))
            return newasid;
        asid &= ~ASID_MASK;
        if (!__test_and_set_bit(asid, asid_map))
            return newasid;
    }


    asid = find_next_zero_bit(asid_map, NUM_USER_ASIDS, cur_idx); - - - （2）
    if (asid != NUM_USER_ASIDS)
        goto set_asid;

    generation = atomic64_add_return_relaxed(ASID_FIRST_VERSION, - - - - （3）
                        &asid_generation);
    flush_context(cpu);

    asid = find_next_zero_bit(asid_map, NUM_USER_ASIDS, 1); - - - - - - （4）

set_asid:
    __set_bit(asid, asid_map);
    cur_idx = asid;
    return asid | generation;
}
```

（1）在创建新的进程的时候会分配一个新的mm，其software asid（mm->context.id）初始化为0。如果asid不等于0那么说明这个mm之前就已经分配过software asid（generation + hw asid）了，那么new context不过就是将software asid中的旧的generation更新为当前的generation而已。

（2）如果asid等于0，说明我们的确是需要分配一个新的HW asid，这时候首先要找一个空闲的HW asid，如果能够找到（jump to set_asid），那么直接返回software asid（当前generation + 新分配的hw asid）。

（3）如果找不到一个空闲的HW asid，说明HW asid已经用光了，这是只能提升generation了。这时候，多有cpu上的所有的old generation需要被flush掉，因为系统已经准备进入new generation了。顺便一提的是这里generation变量已经被赋值为new generation了。

（4）在flush_context函数中，控制HW asid的asid_map已经被全部清零了，因此，这里进行的是new generation中HW asid的分配。

3、进程切换过程中ARM64的tlb操作以及ASID的处理

代码位于arch/arm64/mm/context.c中的check_and_switch_context：

```
void check_and_switch_context(struct mm_struct *mm, unsigned int cpu)
{
    unsigned long flags;
    u64 asid;

    asid = atomic64_read(&mm->context.id);  - - - - - - - - - - - - - - （1）

    if (!((asid ^ atomic64_read(&asid_generation)) >> asid_bits)  - - - - - - （2）
        && atomic64_xchg_relaxed(&per_cpu(active_asids, cpu), asid))
        goto switch_mm_fastpath;

    raw_spin_lock_irqsave(&cpu_asid_lock, flags);
    asid = atomic64_read(&mm->context.id);
    if ((asid ^ atomic64_read(&asid_generation)) >> asid_bits) {  - - - - - - （3）
        asid = new_context(mm, cpu);
        atomic64_set(&mm->context.id, asid);
    }

    if (cpumask_test_and_clear_cpu(cpu, &tlb_flush_pending))  - - - - - - （4）
        local_flush_tlb_all();

    atomic64_set(&per_cpu(active_asids, cpu), asid);
    raw_spin_unlock_irqrestore(&cpu_asid_lock, flags);

switch_mm_fastpath:
    cpu_switch_mm(mm->pgd, mm);
}
```

看到这些代码的时候，你一定很抓狂：本来期望支持ASID的情况下，进程切换不需要TLB flush的操作了吗？怎么会有那么多代码？呵呵～～实际上理想很美好，现实很骨干，代码中嵌入太多管理asid的内容了。

（1）现在准备切入mm变量指向的地址空间，首先通过内存描述符获取该地址空间的ID（software asid）。需要说明的是这个ID并不是HW asid，实际上mm->context.id是64个bit，其中低16 bit对应HW 的ASID（ARM64支持8bit或者16bit的ASID，但是这里假设当前系统的ASID是16bit）。其余的bit都是软件扩展的，我们称之generation。

（2）arm64支持ASID的概念，理论上进程切换不需要TLB的操作，不过由于HW asid的编址空间有限，因此我们扩展了64 bit的software asid，其中一部分对应HW asid，另外一部分被称为asid generation。asid generation从ASID_FIRST_VERSION开始，每当HW asid溢出后，asid generation会累加。asid_bits就是硬件支持的ASID的bit数目，8或者16，通过ID_AA64MMFR0_EL1寄存器可以获得该具体的bit数目。

当要切入的mm的software asid仍然处于当前这一批次（generation）的ASID的时候，切换中不需要任何的TLB操作，可以直接调用cpu_switch_mm进行地址空间的切换，当然，也会顺便设定active_asids这个percpu变量。

（3）如果要切入的进程和当前的asid generation不一致，那么说明该地址空间需要一个新的software asid了，更准确的说是需要推进到new generation了。因此这里调用new_context分配一个新的context ID，并设定到mm->context.id中。

（4）各个cpu在切入新一代的asid空间的时候会调用local_flush_tlb_all将本地tlb flush掉。

参考文献：

1、64-ia-32-architectures-software-developer-manual-325462.pdf

2、DDI0487A_e_armv8_arm.pdf

3、Linux 4.4.6内核源代码

原创文章，转发请注明出处。蜗窝科技

标签: TLB 进程切换

« ARM Linux上的系统调用代码分析 | eMMC 原理 3： 分区管理»

评论：

菜鸟学习linux
2020-09-17 11:48

很棒，虽然没看过代码，但基本上也了解了linux在进程切换mmu页表是如何切换的。。。
有个问题，kernel可以借用用户的mmu页表，那么硬件上是如何控制用户态访问不到kernel空间的

回复

futureland
2019-08-28 10:12

楼主请教个问题：
借用4.1的图，TLB的维护由软件触发，那必然会有延时。如果A0修改了映射关系，在A2中的TLB缓存还没有被flush时，A2会继续维护之前的映射关系，我理解在这种前提下有以下几种情况：
1. A0 unmap 一段空间[1,2]，这时A2继续访问[1,2]会访问到无效的page而并不会造成崩溃；
2. A0和A2如果都在内核态，通过vmalloc相关的系列操作，也应该有以上的情况。

不知道这些情况是否真实存在，内核中是如何处理？

回复

菜鸟学习linux
2020-09-17 11:44

@futureland：“A2继续访问[1,2]会访问到无效的page而并不会造成崩溃”，理解有问题吧，a2的tlb的缓存不invalid的话，A2访问[1,2]会tlb hit，这样会造成，A2的查询到的地址映射还是之前未修改的，修改过的映射还躺在内存里。

回复

七爷不舔屁眼
2019-07-08 16:16

完美解决疑问，非常感谢！

回复

初学者
2018-10-29 12:53

这边不是很懂

原文: "在x86支持PCID（X86术语，相当与ARM的ASID）的情况下会怎样呢？也会在load cr3的时候flush掉所有的本地CPU上的 local tlb entry吗？其实在linux中，由于TLB shootdown，普通的linux并不支持PCID（KVM中会使用，但是不在本文考虑范围内），因此，对于x86的进程地址空间切换，它就是会有flush local tlb entry这样的side effect。"

所以是说x86在多核情况下支持PCID，但是因为TLB shootdown，所以还是需要flush local tlb entry (透过IPI)。单核就不会有side effect，这样吗？

回复

初学者
2018-10-29 13:22

@初学者：另外是，在多核情况下，看起来不管有没有支持PCID，都会有TLB shootdown，所以PCID在多核情况下是没有什么明显的好处的吗？

回复

lanxinyuchs
2018-03-02 14:29

"如果要切入的next task B是内核线程，那么我们也暂时不需要flush TLB，因为内核线程不会访问usersapce"那如果是内核要执行copy_from_user呢，也不需要访问userspace的地址吗，就是小于0xC0000000的虚拟地址部分？

回复

linuxer
2018-03-03 22:54

@lanxinyuchs：内核线程没有用户空间，怎么执行copy_from_user？

回复

crystal736
2017-12-26 18:09

有个问题请教下：
在缺页异常do_page_fault函数中会分配页面，如果分配页面失败会怎么办呢？在do_anonymous_page函数中分配内存是调用alloc_page(GFP_HIGHUSER);而GFP_HIGHUSER包含____GFP_WAIT,是不是分配失败会进入睡眠状态或者一直在waiting？但是在异常处理函数里面一直waiting不太合适吧？这一点想不明白，还请指教

回复

hello-world
2017-12-27 19:30

@crystal736：在缺页异常的处理过程中，如果分配page frame失败，那么进程会进入sleep状态，之后在适当的时间（例如页面回收之后有充足的free page）唤醒该进程。

回复

crystal736
2017-12-25 11:37

一直有一个问题困扰我，我看的是mips架构相关的linux代码。在用户访问未映射的虚拟地址时会出现page fault，tlb miss会进行tlb重填(mips是tlb软件充填)，但是在充填过程中会访问页表，如果此时页表未建立，不是又发生一次page fault？页表是放在mips的kseg哪个区，页表自身需要映射吗？还有在get_free_pages返回的地址应该是在哪个kseg区？

回复

小学生
2017-06-10 18:24

楼主，对于这句：“答案是在下一次进程切换中。因为一旦内核线程被schedule out，并且切入一个新的进程C，那么在switch_mm，切入到C进程地址空间的时候，所有之前的残留都会被清除掉（因为有load cr3的动作）”，如果切入的进程还是A task呢，因为这时候不会flush tlb，那之前其他CPU修改过了的地址映射关系什么时候在当前CPU上体现呢？是根据cpu_tlbstate.state==TLBSTATE_LAZY来判断是否需要flush吗？

回复

linuxer
2017-06-12 17:33

@小学生：如果切入的进程还是A task呢？

这时候进程切换的时候仍然会调用switch_mm(oldmm, mm, next);当然，oldmm和mm是相同的。你可以看看X86如何实现这个函数的，答案也就自然出来了。

回复

chappie
2017-05-06 20:32

我是在《深入理解计算机系统》这本书看到TLB缓存是虚拟寻址的，然后想确认下进程间切换要切出TLB。然后就谷歌到这里来了。题主完美解决了我的疑问。非常赞！！！

回复

温柔海洋
2017-02-10 08:54

虽然没时间看，但是也要很赞一下。

回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论