蜗窝科技

慢下来，享受技术。

进程切换分析(1)：基本框架

作者：linuxer 发布于：2017-2-4 18:43 分类：进程管理

一、前言

本文主要是以context_switch为起点，分析了整个进程切换过程中的基本操作和基本的代码框架，很多细节，例如tlb的操作，cache的操作，锁的操作等等会在其他专门的文档中描述。进程切换包括体系结构相关的代码和系统结构无关的代码。第二、三、四分别描述了context_switch的代码脉络，后面的章节是以ARM64为例子，讲述了具体进程地址空间的切换过程和硬件上下文的切换过程。

二、context_switch代码分析

在kernel/sched/core.c中有一个context_switch函数，该函数用来完成具体的进程切换，代码如下（本文主要描述进程切换的基本逻辑，因此部分代码会有删节）：

```
static inline struct rq * context_switch(struct rq *rq, struct task_struct *prev,
        struct task_struct *next) - - - - - - - - - - - - - - - - - - - (1)
{
    struct mm_struct *mm, *oldmm;

    mm = next->mm;
    oldmm = prev->active_mm; - - - - - - - - - - - - - - - - - - - - (2)

    if (!mm) { - - - - - - - - - - - - - - - - - - - - - - - - - - - - (3)
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next); - - - - - - - - - - - - - - - - - (4)
    } else
        switch_mm(oldmm, mm, next); - - - - - - - - - - - - - - - - - (5)

    if (!prev->mm) { - - - - - - - - - - - - - - - - - - - - - - - - - (6)
        prev->active_mm = NULL;
        rq->prev_mm = oldmm;
    }

    switch_to(prev, next, prev); - - - - - - - - - - - - - - - - - - - (7)
    barrier();

    return finish_task_switch(prev);
}
```

（1）一旦调度器算法确定了pre task和next task，那么就可以调用context_switch函数实际执行进行切换的工作了，这里我们先看看参数传递情况：

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

rq：在多核系统中，进程切换总是发生在各个cpu core上，参数rq指向本次切换发生的那个cpu对应的run queue
prev：将要被剥夺执行权利的那个进程
next：被选择在该cpu上执行的那个进程

（2）next是马上就要被切入的进程（后面简称B进程），prev是马上就要被剥夺执行权利的进程（后面简称A进程）。mm变量指向B进程的地址空间描述符，oldmm变量指向A进程的当前正在使用的地址空间描述符（active_mm）。对于normal进程，其任务描述符（task_struct）的mm和active_mm相同，都是指向其进程地址空间。对于内核线程而言，其task_struct的mm成员为NULL（内核线程没有进程地址空间），但是，内核线程被调度执行的时候，总是需要一个进程地址空间，而active_mm就是指向它借用的那个进程地址空间。

（3）mm为空的话，说明B进程是内核线程，这时候，只能借用A进程当前正在使用的那个地址空间（prev->active_mm）。注意：这里不能借用A进程的地址空间（prev->mm），因为A进程也可能是一个内核线程，不拥有自己的地址空间描述符。

（4）如果要切入的B进程是内核线程，那么调用体系结构相关的代码enter_lazy_tlb，标识该cpu进入lazy tlb mode。那么什么是lazy tlb mode呢？如果要切入的进程实际上是内核线程，那么我们也暂时不需要flush TLB，因为内核线程不会访问usersapce，所以那些无效的TLB entry也不会影响内核线程的执行。在这种情况下，为了性能，我们会进入lazy tlb mode。进程切换中和TLB相关的内容我们会单独在一篇文章中描述，这里就不再赘述了。

（5）如果要切入的B进程是内核线程，那么由于是借用当前正在使用的地址空间，因此没有必要调用switch_mm进行地址空间切换，只有要切入的B进程是一个普通进程的情况下（有自己的地址空间）才会调用switch_mm，真正执行地址空间切换。

如果切入的是普通进程，那么这时候进程的地址空间已经切换了，也就是说在A--->B进程的过程中，进程本身尚未切换，而进程的地址空间已经切换到了B进程了。这样会不会造成问题呢?还好，呵呵，这时候代码执行在kernel space，A和B进程的kernel space都是一样一样的啊，即便是切了进程地址空间，不过内核空间实际上保持不变的。

（6）如果切出的A进程是内核线程，那么其借用的那个地址空间（active_mm）已经不需要继续使用了（内核线程A被挂起了，根本不需要地址空间了）。除此之外，我们这里还设定了run queue上一次使用的mm struct（rq->prev_mm）为oldmm。为何要这么做？先等一等，下面我们会统一描述。

（7）一次进程切换，表面上看起来涉及两个进程，实际上涉及到了三个进程。switch_to是一个有魔力的符号，和一般的调用函数不同，当A进程在CPUa调用它切换到B进程的时候，switch_to一去不回，直到在某个cpu上（我们称之CPUx）完成从X进程（就是last进程）到A进程切换的时候，switch_to返回到A进程的现场。这一点我们会在下一节详细描述。switch_to完成了具体prev到next进程的切换，当switch_to返回的时候，说明A进程再次被调度执行了。

三、switch_to为什么需要三个参数呢?

switch_to定义如下：

```
#define switch_to(prev, next, last)                     \
    do {                                    \
        ((last) = __switch_to((prev), (next)));            \
    } while (0)
```

一个switch_to将代码分成两段：

```
AAA

switch_to(prev, next, prev);

BBB
```

一次进程切换，涉及到了三个进程，prev和next是大家都熟悉的参数了，对于进程A（下图中的右半图片），如果它想要切换到B进程，那么：
prev = A

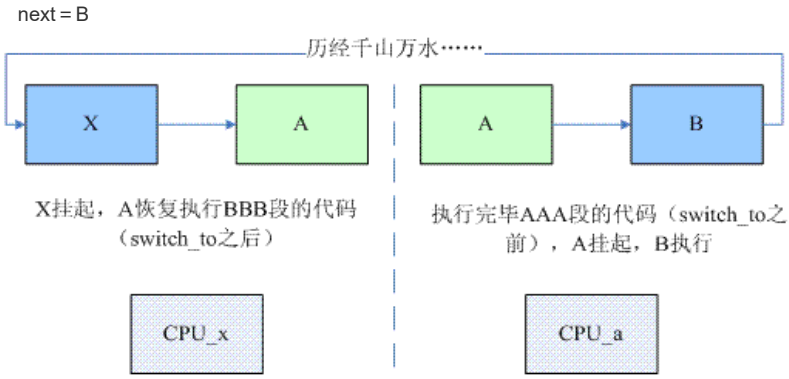
项目专区(0)
X Project(28)

随机文章

Linux TTY framework(3)_从应用的角度看TTY设备
linux 串口调试方法
CFS调度器（3）-组调度
ARM64的启动过程之（二）：创建启动阶段的页表
kernel启动优化

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)
2015年4月(9)



这时候，在A进程中调用 switch_to 完成A到B进程的切换。但是，当经历万水千山，A进程又被重新调度的时候，我们又来到了switch_to返回的这一点（下图中的左半图片），这时候，我们是从哪一个进程切换到A呢？谁知道呢（在A进程调用switch_to 的时候是不知道的）？在A进程调用switch_to之后，cpu就执行B进程了，后续B进程切到哪个进程呢？随后又经历了怎样的进程切换过程呢？当然，这一切对于A进程来说它并不关心，它唯一关心的是当切换回A进程的时候，该cpu上（也不一定是A调用switch_to切换到B进程的那个CPU）执行的上一个task是谁？这就是第三个参数的含义（实际上这个参数的名字就是last，也基本说明了其含义）。也就是说，在AAA点上，prev是A进程，对应的run queue是CPUa的run queue，而在BBB点上，A进程恢复执行，last是X进程，对应的run queue是CPUx的run queue。

四、在内核线程切换过程中，内存描述符的处理

我们上面已经说过：如果切入内核线程，那么其实进程地址空间实际上并没有切换，该内核线程只是借用了切出进程使用的那个地址空间（active_mm）。对于内核中的实体，我们都会使用引用计数来根据一个数据对象，从而确保在没有任何引用的情况下释放该数据对象实体，对于内存描述符亦然。因此，在context_switch中有代码如下：

```
if (!mm) {
    next->active_mm = oldmm;
    atomic_inc(&oldmm->mm_count); - - - - - 增加引用计数
    enter_lazy_tlb(oldmm, next);
}
```

既然是借用别人的内存描述符（地址空间），那么调用atomic_inc是合理的，反正马上就切入B进程了，在A进程中提前增加引用计数也OK的。话说有借有还，那么在内核线程被切出的时候，就是归还内存描述符的时候了。

这里还有一个悖论，对于内核线程而言，在运行的时候，它会借用其他进程的地址空间，因此，在整个内核线程运行过程中，需要使用该地址空间（内存描述符），因此对内存描述符的增加和减少引用计数的操作只能在在内核线程之外完成。假如一次切换是这样的：…A--->B（内核线程）--->C…，增加引用计数比较简单，上面已经说了，在A进程调用context_switch的时候完成。现在问题来了，如何在C中完成减少引用计数的操作呢？我们还是从代码中寻找答案，如下（context_switch函数中，去掉了不相关的代码）：

```
if (!prev->mm) {
    prev->active_mm = NULL;
    rq->prev_mm = oldmm; - - - 在rq->prev_mm上保存了上一次使用的mm struct
}
```

借助其他进程内存描述符的东风，内核线程B欢快的运行，然而，快乐总是短暂的，也许是B自愿的，也许是强迫的，调度器最终会剥夺B的执行，切入C进程。也就是说，B内核线程调用switch_to（执行了AAA段代码），自己挂起，C粉墨登场，执行BBB段的代码。具体的代码在finish_task_switch，如下：

```
static struct rq *finish_task_switch(struct task_struct *prev)
{
    struct rq *rq = this_rq();
    struct mm_struct *mm = rq->prev_mm;————————————————————（1）
```

```
    rq->prev_mm = NULL;

    if (mm)
        mmdrop(mm);————————————————————————————————（2）
}
```

（1）我们假设B是内核线程，在进程A调用context_switch切换到B线程的时候，借用的地址空间被保存在CPU对应的run queue中。在B切换到C之后，通过rq->prev_mm就可以得到借用的内存描述符。

（2）已经完成B到C的切换后，借用的地址空间可以返还了。因此在C进程中调用mmdrop来完成这一动作。很神奇，在A进程中为内核线程B借用地址空间，但却在C进程中释放它。

五、ARM64的进程地址空间切换

对于ARM64这个cpu arch，每一个cpu core都有两个寄存器来指示当前运行在该CPU core上的进程（线程）实体的地址空间。这两个寄存器分别是ttbr0_el1（用户地址空间）和ttbr1_el1（内核地址空间）。由于所有的进程共享内核地址空间，因此所谓地址空间切换也就是切换ttbr0_el1而已。地址空间听起来很抽象，实际上就是内存中的若干Translation table而已，每一个进程都有自己独立的一组用于翻译用户空间虚拟地址的Translation table，这些信息保存在内存描述符中，具体位于struct mm_struct中的pgd成员中。以pgd为起点，可以遍历该内存描述符的所有用户地址空间的Translation table。具体代码如下：

```
static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next,
      struct task_struct *tsk) - - - - - - - - - - - - - - - - （1）
{
    unsigned int cpu = smp_processor_id();

    if (prev == next) - - - - - - - - - - - - - - - - - - - - （2）
        return;

    if (next == &init_mm) { - - - - - - - - - - - - - - - - - （3）
        cpu_set_reserved_ttbr0();
        return;
    }

    check_and_switch_context(next, cpu);
}
```

（1）prev是要切出的地址空间，next是要切入的地址空间，tsk是将要切入的进程。

（2）要切出的地址空间和要切入的地址空间是一个地址空间的话，那么切换地址空间也就没有什么意义了。

（3）在ARM64中，地址空间的切换主要是切换ttbr0_el1，对于swapper进程的地址空间，其用户空间没有任何的mapping，而如果要切入的地址空间就是swapper进程的地址空间的时候，将（设定ttbr0_el1指向empty_zero_page）。

（4）check_and_switch_context中有很多TLB、ASID相关的操作，我们将会在另外的文档中给出细致描述，这里就简单略过，实际上，最终该函数会调用arch/arm64/mm/proc.S文件中的cpu_do_switch_mm将要切入进程的L0 Translation table物理地址（保存在内存描述符的pgd成员）写入ttbr0_el1。

六、ARM64的的进程切换

由于存在MMU，内存中可以有多个task，并且由调度器依次调度到cpu core上实际执行。系统有多少个cpu core就可以有多少个进程（线程）同时执行。即便是对于一个特定的cpu core，调度器可以可以不断的将控制权从一个task切换到另外一个task上。实际的context switch的动作也不复杂：就是将当前的上下文保存在内存中，然后从内存中恢复另外一个task的上下文。对于ARM64而言，context包括：

（1）通用寄存器

（2）浮点寄存器

（3）地址空间寄存器（ttbr0_el1和ttbr1_el1），上一节已经描述

（4）其他寄存器（ASID、thread process ID register等）

__switch_to代码（位于arch/arm64/kernel/process.c）如下：

```
struct task_struct *__switch_to(struct task_struct *prev,
                struct task_struct *next)
{
    struct task_struct *last;

    fpsimd_thread_switch(next); - - - - - - - - - - - - - - - （1）
    tls_thread_switch(next); - - - - - - - - - - - - - - - - - （2）
    hw_breakpoint_thread_switch(next); - - 和硬件跟踪相关
    contextidr_thread_switch(next);  - - 和硬件跟踪相关

    dsb(ish);
    last = cpu_switch_to(prev, next);  - - - - - - - - - - - - （3）

    return last;
}
```

（1）fp是float-point的意思，和浮点运算相关。simd是Single Instruction Multiple Data的意思，和多媒体以及信号处理相关。fpsimd_thread_switch其实就是把当前FPSIMD的状态保存到了内存中（task.thread.fpsimd_state），从要切入的next进程描述符中获取FPSIMD状态，并加载到CPU上。

（2）概念同上，不过是处理tls（thread local storage）的切换。这里硬件寄存器涉及tpidr_el0和tpidrro_el0，涉及的内存是task.thread.tp_value。具体的应用场景是和线程库相关，具体大家可以自行学习了。

（3）具体的切换发生在arch/arm64/kernel/entry.S文件中的cpu_switch_to，代码如下：

```
ENTRY(cpu_switch_to) - - - - - - - - - - - - - - - - - - - - （1）
    mov    x10, #THREAD_CPU_CONTEXT - - - - - - - - - - （2）
    add    x8, x0, x10 - - - - - - - - - - - - - - - - - - - - （3）
    mov    x9, sp
    stp    x19, x20, [x8], #16 - - - - - - - - - - - - - - - - （4）
    stp    x21, x22, [x8], #16
    stp    x23, x24, [x8], #16
    stp    x25, x26, [x8], #16
    stp    x27, x28, [x8], #16
    stp    x29, x9, [x8], #16
    str    lr, [x8] - - - - - - - - - A
    add    x8, x1, x10 - - - - - - - - - - - - - - - - - - - - （5）
    ldp    x19, x20, [x8], #16 - - - - - - - - - - - - - - - - （6）
    ldp    x21, x22, [x8], #16
    ldp    x23, x24, [x8], #16
    ldp    x25, x26, [x8], #16
    ldp    x27, x28, [x8], #16
    ldp    x29, x9, [x8], #16
    ldr    lr, [x8] - - - - - - - B
    mov    sp, x9 - - - - - - - C
    ret - - - - - - - - - - - - - - - - - - - - - - - - - - - - （7）
ENDPROC(cpu_switch_to)
```

（1）进入cpu_switch_to函数之前，x0，x1用做参数传递，x0是prev task，就是那个要挂起的task，x1是next task，就是马上要切入的task。cpu_switch_to和其他的普通函数没有什么不同，尽管会走遍万水千山，但是最终还是会返回调用者函数__switch_to。

在进入细节之前，先想一想这个问题：cpu_switch_to要如何保存现场？要保存那些通用寄存器呢？其实上一小段描述已经做了铺陈：尽管有点怪异，本质上cpu_switch_to仍然是一个普通函数，需要符合ARM64标准过程调用文档。在该文档中规定，x19～x28是属于callee-saved registers，也就是说，在__switch_to函数调用cpu_switch_to函数这个过程中，cpu_switch_to函数要保证x19～x28这些寄存器值是和调用cpu_switch_to函数之前一模一样的。除此之外，pc、sp、fp当然也是必须是属于现场的一部分的。

（2）得到THREAD_CPU_CONTEXT的偏移，保存在x10中

（3）x0是pre task的进程描述符，加上偏移之后就获取了访问cpu context内存的指针（x8寄存器）。所有context的切换的原理都是一样的，就是把当前cpu寄存器保存在内存中，这里的内存是在进程描述符中的 thread.cpu_context中。

（4）一旦定位到保存cpu context（各种通用寄存器）的内存，那么使用stp保存硬件现场。这里x29就是fp（frame pointer），x9保存了stack pointer，lr是返回的PC值。到A代码处，完成了pre task cpu context的保存动作。

（5）和步骤（3）类似，只不过是针对next task而言的。这时候x8指向了next task的cpu context。

（6）和步骤（4）类似，不同的是这里的操作是恢复next task的cpu context。执行到代码B处，所有的寄存器都已经恢复，除了PC和SP，其中PC保存在了lr（x30）中，而sp保存在了x9中。在代码C出恢复了sp值，这时候万事俱备，只等PC操作了。

（7）ret指令其实就是把x30（lr）寄存器的值加载到PC，至此现场完全恢复到调用cpu_switch_to那一点上了。

参考文献：

1、ARM标准过程调用文档（IHI0056C_beta_aaelf64.pdf）

2、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: 进程切换 context_switch

« eMMC 原理 3： 分区管理 | 为什么会有文件系统(一)»

评论：

demowang
2021-09-23 11:59

请教进程的ASID存入ttbr1_el1后，ttbr1_el1如何做内核寻址

回复

qielan
2020-12-24 18:58

@linuxer 线程是调度的单位，为什么大家在说的时候都只是说进程的调度，这只是一种口头习惯还是有一些差别

回复

jack
2020-12-21 09:55

请教一下，进程切换的时候为什么要在rq数据结构中引入prev_mm，
为什么在切换之前现将prev->active_mm保存到rq->prev_mm?
在finish_task_switch()函数中也是可以获取到prev->active_mm，
在此函数中再将prev进程与借用的地址空间断开会引发问题吗?

回复

leon
2021-10-14 21:26

@jack：我的理解是:在finish_task_switch里面获取到的prev已经和上面的prev不是同一个了

回复

伊斯科明
2021-10-15 16:49

@leon：看了@jack的问题，我也有这个疑问。@leon 的说法感觉解释不通啊，因为C进程执行到finish_task_switch后，此时的prev指的就是B进程，这时候确实可以通过prev->active_mm来释放吧

回复

bill
2020-05-06 12:37

请问一下switch_to之后获取到last有啥用啊

回复

bill
2020-05-07 17:39

@bill：知道了，finish_task_switch要用

回复

navadoo
2017-11-21 17:41

请教一下，在很多平台上(包括arm/arm64/x86)，enter_lazy_tlb都是空函数，这个怎么理解。

回复

bsp
2021-01-04 14:09

@navadoo：arm/arm64等将内核空间和用户空间 通过不同页表基地址 隔离开来的架构，由于内核是所有task共享的，其实在切换内核线程时 是不需要要切换内核空间的页表基地址(ttbr1_el1)，所以enter_lazy_tlb可以是空函数。
但如果在arm中enable了SW_TTBR0_PAN（privilege access non，禁止kernel访问用户空间，如需访问使用copy_from/to_user来先disable PAN，访问后再enable），update_saved_ttbr0其实就是将用户空间基地址ttrb0_el1设成zero page（在访问user address时都是得到0）。

```
static inline void
enter_lazy_tlb(struct mm_struct *mm, struct task_struct *tsk)
{
	/*
	 * We don't actually care about the ttbr0 mapping, so point it at the
	 * zero page.
	 */
	update_saved_ttbr0(tsk, &init_mm);
}
```

回复

fanzhang
2017-11-18 05:51

借助其他进程内存描述符的东风，内核线程B欢快的运行，然而，快乐总是短暂的，也许是B自愿的，也许是强迫的，调度器最终会剥夺B的执行，切入C进程。也就是说，B内核线程调用switch_to（执行了AAA段代码），自己挂起，C粉墨登场，执行BBB段的代码。

sorry ,question here

这里最后完成switch_to后面BBB部分 的还是B线程 跟C没有关系

回复

walentine
2020-12-29 16:16

@fanzhang：我也认为有这个问题：A->B后，A先执行switch_to之前的AAA部分代码，切换到B后，自然是B来执行swtich_to后的BBB部分代码，为何B欢快执行后，再走调度切换流程B->C后，要用C来释放借用的A的mm_struct？B和C都是同样的被调度方，B为何不执行finish_task_switch来释放A的mm_struct？

回复

bsp
2021-01-04 16:30

@walentine："这里最后完成switch_to后面BBB部分 的还是B线程"，这句话没有问题。
但是在 "用户线程A->kernel线程B->用户线程C" 这样的例子中，B会mmgrab(A->active_mm)，因为B没有用户地址空间 不需要更新mm，所以B在执行finish_task_switch时并不会mmdrop(A->active_mm);
B切换到C后，如果C是内核线程处理非常简单（C->active_mm = B->active_mm，即A->active_mm），但是如果是用户线程，C在执行finish_task_switch时就会mmdrop(A->active_mm)（说明A没有其它task借用了可以自己玩耍），然后更新C的mm到ttbr0_el1.

回复

bsp
2021-01-04 16:47

@bsp：也可以得出，在这样的例子中：用户线程A->kernel线程B->kernel线程C/D/E/F->用户线程G，只有最后一个用户线程在执行finish_task_switch时才会mmdrop(A->active_mm)并跟新mm，其它内核线程都是在沿用第一个用户线程的active_mm。由此看来，内核线程的切换loading还是比较轻的。

回复

fanzhang
2017-11-18 05:50

借助其他进程内存描述符的东风，内核线程B欢快的运行，然而，快乐总是短暂的，也许是B自愿的，也许是强迫的，调度器最终会剥夺B的执行，切入C进程。也就是说，B内核线程调用switch_to（执行了AAA段代码），自己挂起，C粉墨登场，执行BBB段的代码。

sorry , question
这里最后完成switch_to后面BBB部分 的还是B线程 跟C没有关系

回复

codingbelief
2017-02-05 17:00

“（3）mm为空的话，说明B进程是内核线程，这时候，只能借用A进程当前正在使用的那个地址空间” 这个有点不理解的地方，内核线程是在什么场景下会用到用户线程的地址空间么?

简单看代码的话，这个“借用”貌似只是做了一个传递的动作，最终达到的效果是延迟了备份最后切出去的用户线程 context 的操作，如果没有切换其他用户进程又切回来了，就省了不少事。

回复

linuxer
2017-02-06 10:19

@codingbelief：我们假设场景是这样的：普通进程A切换到内核线程B。这种场景下需要切换进程空间吗?

当然不用，A进程的4G地址空间中（32位ARM为例），3～4G那一段就是对应的内核地址空间，B内核线程没有userspace，没有自己的0～3G那段地址空间，而3～4G那一段的地址空间是和A进程完全一样的，因此，在这样的场景中，不需要切换mm，不需要切换MMU中指向各个level的translation tables的那个基址寄存器（例如X86处理器就是cr3寄存器，ARM64就是TTBRx_EL1）。B线程借用A进程的地址空间，借用的也只是3～4G那一段的页表，对于进程自己特定的0～3G那段地址空间，B是不会访问的。

这种情况下，我们称B线程借用了A进程的地址空间，在B的上下文中，基址寄存器指向A进程的pgd，而那一段保存A进程页表的memory是不属于B进程，因此，我们称之为“借用”。当然，借用不仅仅是“传递”那个动作，最重要的是“增加引用计数”这个动作，这个动作可以确保在B线程借用的过程中，A进程的mm struct是不会被释放的。当然，对于A--->B--->A这样的场景，的确是省事了，而在A--->B--->C（c是普通进程，和A不同地址空间）这样的场景，的确是推迟了实际的地址空间的切换动作。

回复

codingbelief
2017-02-06 16:46

@linuxer：在 ARM64 中，用户进程的 mm->pgd 最终是写入到 TTBR0_EL1，应该可以说该 mm 只涉及了用户空间的虚拟地址。内核空间的虚拟地址的 pgd 应该是 swapper_pg_dir，在内核初始化的时候写入到 TTBR1_EL1，在进程切换过程中是不会改变的，所以内核线程没有“用”到 mm 里面的 pgd。
如果内核线程不会去访问用户空间的话，应该就是不需要“用”到用户进程的 mm。（不知道内核线程有没有场景会去访问用户空间，个人觉得内核线程访问用户空间不是很合理，假定是没有这类场景的。）

“增加引用计数”这个动作，我认为是为了保证“传递”的有效性而进行的。mm 会随着用户进程创建的时候被创建，创建之后，由于用户进程的引用，mm 的引用计数应该加1了，在用户进程切出切去时，由于 mm 被内核进程引用，因此 mm 的引用计数再加1。如果用户进程在切出去后，没有被 kill 掉，那么 mm 的引用计数至少等于 2。如果用户进程切出去后，又被 kill 掉了，引用计数就保证了此时 mm 不会立刻被释放 （因为“传递”还没有完成），等待内核线程再次切出去后，mm 才会被释放。

所以，我觉得 “增加引用计数” 可能只是措施，并不是目的。在 ARM64 里，内核线程可能是“暂存”或者“传递” mm，而不是“借用”。

回复

linuxer
2017-02-06 18:11

@codingbelief：的确，在ARM64中，由于分别有TTBR0_EL1 TTBR1_EL1分别指向user和kernel address space，这导致在进程切换中，不需要切换TTBR1_EL1的值，只需要切换TTBR0_EL1的值即可。这时候，如果普通进程A切换到内核线程B，那么B线程的确是不需要“借用”A的mm struct中的pgd，因为对于ARM64而言，mm struct中的pgd表示就是userspace的地址空间。

然而，linux kernel是支持很多cpu arch的，ARM、X86等处理器都不是这么玩的。因此才会有内核

线程“借用”进程地址空间一说，因为大部分的cpu arch都需要“借用”，当然，对于ARM64而言，它根本不需要active mm这样的概念。

回复

codingbelief

2017-02-06 17:54

@linuxer：X -> A -> B 场景中, X A 为用户进程，B 为内核进程，在这种切换场景下，A -> B 时，没有实际进行 mm 切换，是不是可以说切换到 B 时，B 所在的地址空间实际是属于 X 的?

（只是觉得“借用”可能会有点疑惑，别怪我在抠字眼，~ ~）

回复

linuxer

2017-02-06 18:14

@codingbelief：这时候B所在的地址空间应该是属于A的吧，A是普通用户进程啊

回复

codingbelief

2017-02-06 18:17

@codingbelief：囧，这里犯迷糊了，B 所在的地址空间实际是属于 A 的，应该是 A 的 mm 没有被切换出去，所以 B 还是复用 A 的 mm。

回复

linuxer

2017-02-06 18:50

@codingbelief：吃饭归来，就“借用”再说一句吧，呵呵 ~ ~ ~
虽然在ARM64上，内核线程不借用mm，但是在ARM、x86上还是需要“借用”，之所以使用这个词是因为内核线程的的确确是使用了其他进程的页表。

还是使用普通进程A切换到内核线程B这个场景，B线程当然只会访问内核空间的地址了，访问其他进程userspace是非法的。

B线程访问内核空间的地址的时候，无论是取指还是数据操作，都需要进行地址翻译（地址属于内核空间），因此需要找到当前cpu上的那个Translation table base address寄存器，这时候，该寄存器的地址被设定为A进程mm的pgd，因此，在B线程执行，进行地址翻译的时候，顺着Translation table base address寄存器来到了A进程的页表区域，将内核空间的虚拟地址翻译成对应的物理地址。

因为B线程的确是使用了属于A进程的东西（页表），所以我使用了“借用”这个词。

回复

codingbelief

2017-02-06 20:13

@linuxer：arm 和 x86 的内核虚拟地址的页表应该是只有一个独立实例，然后其他所用户进程在他们的页表中，将内核虚拟地址对应的一级页表条目指向内核页表，是这样实现的么?

回复

linuxer

2017-02-06 23:44

@codingbelief：在arm 和 x86的情况下，进程地址空间应该包括：
1、一个pgd，每一个entry指向一个pud。
2、若干个pud，每一个entry指向一个pmd
3、若干个pmd，每一个entry指向一个pte
4、若干个pte，每一个entry指向一个page frame

就kernel space地址空间而言，系统中所有的进程、内核线程都有自己独立的pgd，但是大家会共享PUD、PMD、PTE。

为什么pgd不能共享呢？因为pgd中的部分entry是for kernel space的，部分entry是for userspace的，虽然各个进程（线程）的kernel space的pgd entry是相同的，但是user space各不相同，因此不能共享。

“页表”这个术语我一般指PGD/PUD/PMD/PTD，对于ARM，就是各个level的Translation tables。你说“arm 和 x86 的内核虚拟地址的页表应该是只有一个独立实例”，我大概明白你的意思，不过不是非常的准确。

bsp

2021-01-04 16:48

@linuxer：ttbr1_el1 Is the page table base register in kernel space , Use when accessing kernel space address , All processes share , No need to switch.
kernel threads之间的切换是不需要切换ttrb1_el1的

回复

bsp
2021-01-04 17:04

@bsp：如果为了模拟PAN是需要切换ttbr1的，看来PAN对进程切换的latency影响不小啊。

tiny-laker
2017-02-05 09:35

先顶后看好习惯

回复

demowang
2021-09-23 11:58

@tiny-laker：请教进程的ASID存入ttbr1_el1后，ttbr1_el1如何做内核寻址

回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论