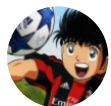


# Linux内核浅析-进程调度时机和过程



Howard

+ 关注他

2 人赞同了该文章

上文浅析了进程调度算法，本文再浅析下进程调度发生的时机以及过程。前文有调度算法和x86体系结构的结构，这两篇文章是本文的基础。

## 1、调度类型和时机

调度触发有两种类型，进程主动触发的主动调度和被动调度，被动调度又叫抢占式调度。

主动调度：进程主动触发以下情况，然后陷入内核态，最终调用schedule函数，进行调度。

- 1、当进程发生需要等待IO的系统调用，如read、write。
- 2、进程主动调用sleep时。
- 3、进程等待占用信用量或mutex时，注意spin锁不会触发调度，可能在空转。

被动调度：当发生以下情况时会发生被动调度：

- 1、tick\_clock，cpu的时钟中断，一般是10ms一次，也有1ms一次的，取决于cpu的主频，此时会通过cfs检查进程队列，如果当前占用cpu的进程的vruntime不是最小时，且超过sched\_min\_granularity\_ns（详细可见前文调度算法），发生“被动调度”，此处有引号，原由下面说。
- 2、fork出新进程时，此时会通过cfs算法检查进度队列，如果当前占用cpu的进程的vruntime不是最小时且超过sched\_min\_granularity\_ns，发生“被动调度”，此处有引号，原由下面说。

为什么上面“被动调度”加引号了？因为被动调度不是立即进行的。上面两种情况仅仅是确认需要调度后给进程的打上标志\_TIF\_NEED\_RESCHED，然后会在以下时机会检查\_TIF\_NEED\_RESCHED标志，如果标志存在再调用schedule函数：

- 1、中断结束返回用户态或内核态之前。
- 2、开启内核抢占开关后。kernal2.5 引入内核抢占，即在内核态也允许抢占。但不是内核态运行全周期都允许去抢占，所以thread\_info.preempt\_count用于标志当前是否可以进行内核抢占。当使用preempt\_enable()开关打开时，会检查\_TIF\_NEED\_RESCHED，进行调度。

从上可以总结下：

- 1、所有调度的发生都是出于内核态，中断也是出于内核态，不会有调度出现在用户态。
- 2、所有调度的都在schedule函数中发生。

## 2、调度代码逻辑

代码调用层次简单提一下，方便需要撸源码的同学理理思路。

```
schedule -> __schedule -> pick_next_task -> fail_sched_class.pick_next_task_fair
-> update_curr, pick_next_entity, context_switch
```

schedule：通过preempt\_disable()首先关闭内核抢占，然后调用\_\_schedule。

\_\_schedule：smp\_processor\_id()获取当前运行的cpu id，rq = cpu\_rq(cpu\_id)，获取当前cpu的调度队列rq（该数据结构可参考前文：[zhuanlan.zhihu.com/p/75...](http://zhuanlan.zhihu.com/p/75...)）

pick\_next\_task：遍历所有调度的sched\_class，并调用sched\_class.pick\_next\_task方法。实时进程的sched\_class在链表前段，会被优先遍历并且调用，以保证实时进程优先被调度。同时本函数进行优化，如果rq -> nr\_running == rq -> cfs.h\_nr\_running，表示队列中的进程数 == cfs调度器中的进程数，即所有进程都是普通进程，则直接使用cfs调度器。ps：pick\_next\_task会完成进程调度，被调度出的进程会在此处暂时结束，当从pick\_next\_task返回的时候已经是下一次再将该进程调入cpu之后才执行，这块会在context\_switch中详细讲。

pick\_next\_task\_fair：如果是公平调度器，则调用fail\_sched\_class.pick\_next\_task\_fair，其包含update\_curr, pick\_next\_entity, context\_switch三个函数。

update\_curr：更新当前进程的vruntime，然后更新红黑树和cfs\_rq -> min\_vruntime以及left\_most。

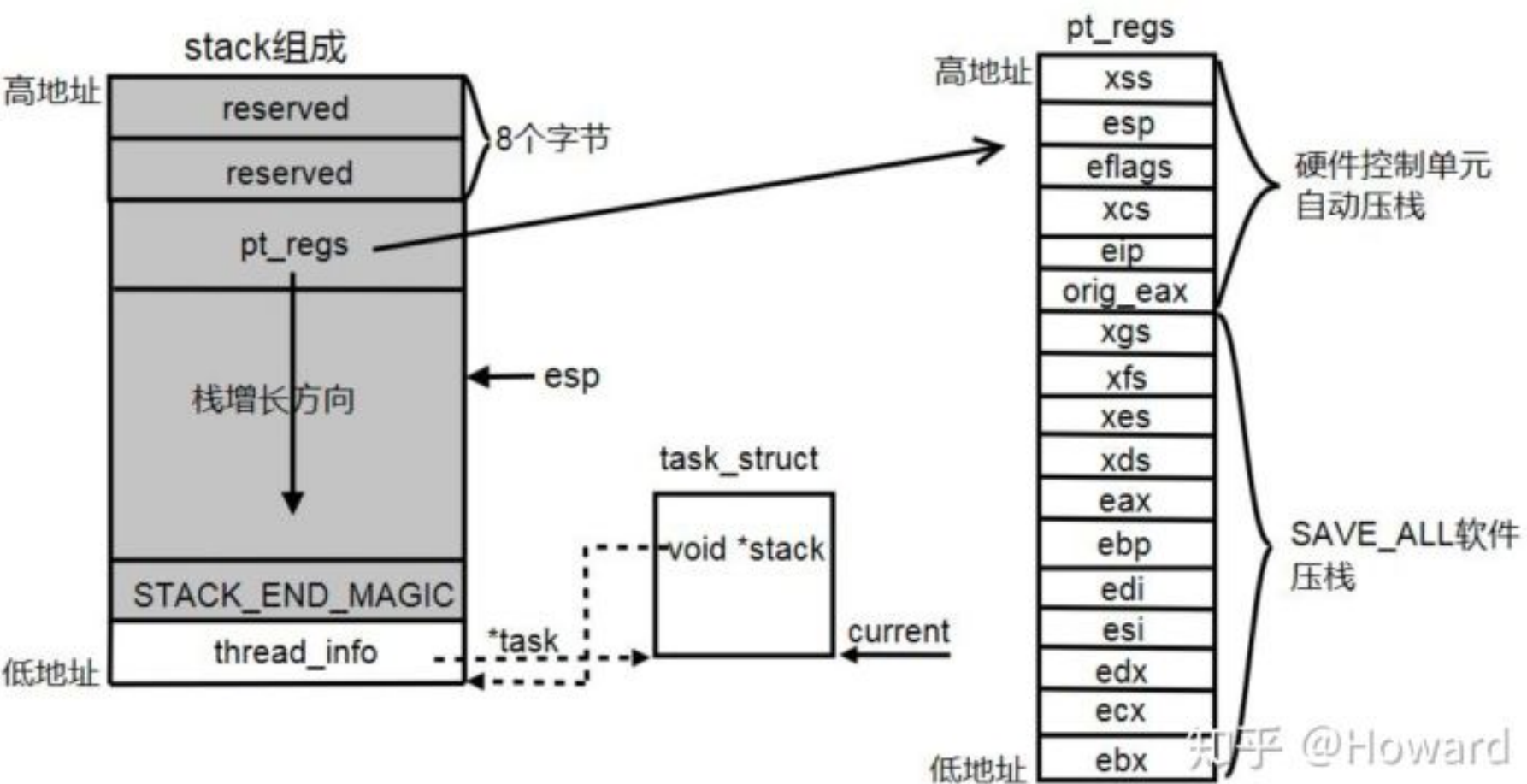
pick\_next\_entity：选择红黑树的left\_most，比较和当前进程和left\_most是否是同一进程，如果不是则进行context\_switch。

### 3、context switch（上下文切换）

这是进程调度最难的部分，因为涉及硬件，linux也会支持不同的硬件体系。不过搞懂了上下文切换，对于硬件和linux会有更深入的了解。

介绍上下文切换前，需要介绍下相关的数据结构：内核栈、thread\_struct、tss。

1、内核栈：进程进入内核态后使用内核栈，和用户栈完全隔离，task\_struct -> stack指向该进程的内核栈，大小一般为8k。



```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

```
};
```

整个内核栈用union表示，thread\_info和stack共用一段存储空间，thread\_info占用低地址。在pt\_regs和STACK\_END\_MAGIC之间，就是内核代码的运行栈。当内核栈增长超过STACK\_END\_MAGIC就会报内核栈溢出。

1) thread\_info：存储内核态运行的一些信息，如指向task\_struct的task指针，使得陷入内核态之后仍然能够找到当前进程的task\_struct，还包括是否允许内核中断的preempt\_count开关等等。

```
struct thread_info {
    unsigned long      flags;          /* low level flags */
    mm_segment_t       addr_limit;     /* address limit */
    struct task_struct  *task;         /* main task structure */
    int                 preempt_count; /* 0 => preemptable, <0 => bug */
    int                 cpu;           /* cpu */
};
```

2) pt\_regs：存储用户态的硬件上下文（**ps：用户态**）。用户态 -> 内核态后，由于使用的栈、内存地址空间、代码段等都不同，所以用户态的eip、esp、ebp等需要保存现场，内核态 -> 用户态时再将栈中的信息恢复到硬件。由于进程调度一定会在内核态的schedule函数，用户态的所有硬件信息都保存在pt\_regs中了。SAVE\_ALL指令就是将用户态的cpu寄存器值保存如内核栈，RESTORE\_ALL就是将pt\_regs中的值恢复到寄存器中，这两个指令在介绍中断的时候还会提到。

3、TSS（task state segment）：这是intel为上层做进程切换提供的硬件支持，还有一个TR（task register）寄存器专门指向这个内存区域。当TR指针值变更时，intel会将当前所有寄存器值存放到当前进程的tss中，然后再讲切换进程的目标tss值加载后寄存器中，其结构如下：

I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

这里很多人都会有疑问，不是有内核栈的pt\_regs存储硬件上下文了吗，为什么还要有tss？前文说过，进程切换都是在内核态，而pt\_regs是保存的用户态的硬件上下文，tss用于保存内核态的硬件上下文。

但是linux并没有买账使用tss，因为linux实现进程切换时并不需要所有寄存器都切换一次，如果使用tr去切换tss就必须切换全部寄存器，性能开销会很高。这也是intel设计的败笔，没有把这个功能做的更加的开放导致linux没有用。linux使用的是软切换，主要使用thread\_struct，tss仅使用esp0这个值，用于进程在用户态 -> 内核态时，硬件会自动将该值填充到esp寄存器。在初始化时仅为每1个cpu仅绑定一个tss，然后tr指针一直指向这个tss，永不切换。

4、thread\_struct：一个和硬件体系强相关的结构体，用来存储内核态切换时的硬件上下文。

```
struct thread_struct {
    unsigned long    rsp0;
    unsigned long    rsp;
    unsigned long    userrsp;    /* Copy from PDA */
    unsigned long    fs;
    unsigned long    gs;
```



```

        unsigned short    es, ds, fsindex, gsindex;
/* Hardware debugging registers */
.....
/* fault info */
        unsigned long    cr2, trap_no, error_code;
/* floating point info */
        union i387_union    i387    __attribute__((aligned(16)));
/* IO permissions. the bitmap could be moved into the GDT, that would make
switch faster for a limited number of ioperm using tasks. -AK */
        int            ioperm;
        unsigned long    *io_bitmap_ptr;
        unsigned io_bitmap_max;
/* cached TLS descriptors. */
        u64 tls_array[GDT_ENTRY_TLS_ENTRIES];
} __attribute__((aligned(16)));

```

5、进程切换逻辑主要分为两部分：1) switch\_mm\_irqs\_off：切换进程内存地址空间，对于每个进程都有一个进程内存地址空间，是一个以进程隔离的虚拟内存地址空间，所以此处也需要切换，包括页表等，后面后详细讲到。2) switch\_to：切换寄存器和堆栈。

```

/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
    struct mm_struct *mm, *oldmm;
    .....
    mm = next->mm;
    oldmm = prev->active_mm;
    .....
    switch_mm_irqs_off(oldmm, mm, next);
    .....
    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, prev);
    barrier();
    return finish_task_switch(prev);
}

```

在switch\_to中直接调用汇编\_\_switch\_to\_asm，进入\_\_switch\_to\_asm前，eax存储prev task（当前进程，即将被换出）的task\_struct指针，edx存储next task（即将被换入的进程）的task\_struct指针。

```

/*
 * %eax: prev task
 * %edx: next task
 */
ENTRY(__switch_to_asm)
    /*
     * Save callee-saved registers
     * This must match the order in struct inactive_task_frame
     */
    pushl    %ebp
    pushl    %ebx
    pushl    %edi
    pushl    %esi

```

```

        pushfl

        /* switch stack */
        movl    %esp, TASK_threadsp(%eax)
        movl    TASK_threadsp(%edx), %esp
    ....
        /* restore callee-saved registers */
        popfl
        popl    %esi
        popl    %edi
        popl    %ebx
        popl    %ebp

        jmp     __switch_to
END(__switch_to_asm)

```

- 1) 将prev task的ebp、ebx、edi、esi、eflags寄存器值压入prev task的内核栈。
- 2) TASK\_threadsp是从task\_struct -> thread\_struct -> sp获取esp指针。在switch stack阶段，首先保存prev task内核栈的esp指针到thread\_struct -> sp。然后将next的thread\_struct -> sp恢复到esp寄存器，此后所有的操作都在next task的内核栈上运行。

只要完成了esp寄存器的切换，基本就完成了进程的切换最核心的一步。因为通过esp找到next task的内核栈，然后就能在内核栈中找到其他寄存器的值（步骤1压入的寄存器值）和通过thread\_info找到task\_struct.thread\_struct。

- 3) 将next task的eflags、esi、edi、ebx、ebp pop到对应的寄存。和步骤1push的顺序正好相反。

\_\_switch\_to:

```

struct task_struct * __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread;
    struct thread_struct *next = &next_p->thread;
    ....
    int cpu = smp_processor_id();
    struct tss_struct *tss = &per_cpu(cpu_tss, cpu);
    ....
    load_TLS(next, cpu);
    ....
    this_cpu_write(current_task, next_p);

    /* Reload esp0 and ss1. This changes current_thread_info(). */
    load_sp0(tss, next);
    ....
    return prev_p;
}

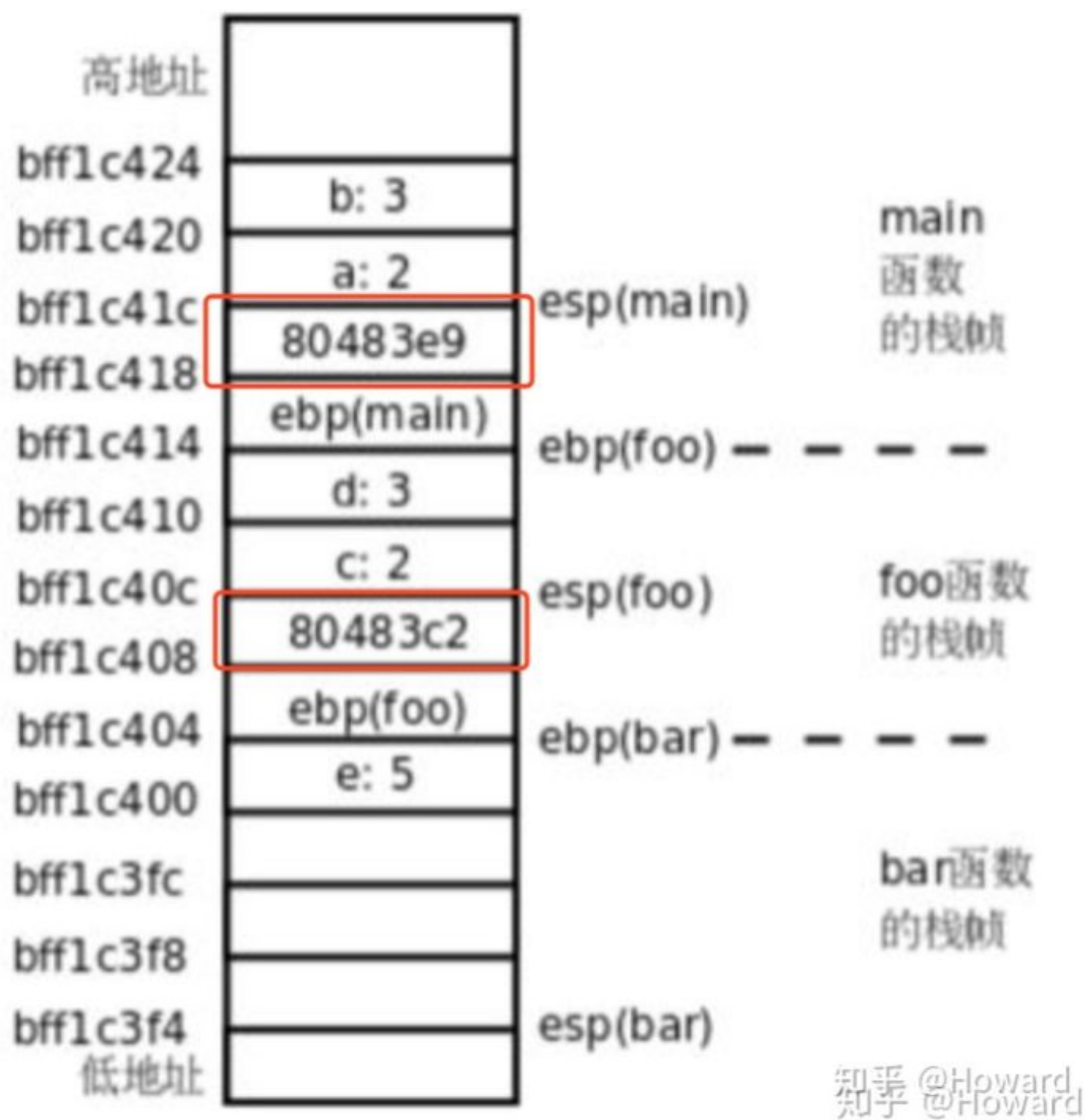
```

- 1) load\_TLS：加载next task的TLS（进程局部变量）到CPU的GDT（全局描述符表，global descriptor table）的TLS中，关于GDT和TLS后面中断的时候会着重讲这两个结构。
- 2) load\_sp0：将next task的esp0加载到tss中。esp和esp0的区别是前者是用户态栈的esp，后者是内核栈的esp。当从用户态进入内核态（ring0优先级）时，硬件会自动将esp = tss -> esp0。切换esp后，再进行弹栈等操作回复其他的寄存器，如switch宏后半部分一样。

内存虚拟空间、寄存器、内核栈都恢复了，还有一个重要的EIP（指令指针寄存器）还没有恢复。但linux的做法是不恢复EIP寄存器。

1) 当prev -> next内核栈完成切换后（假设prev是A进程，next是B进程），EIP仍然指向switch\_to函数，因为A进程是在执行到switch\_to的时候结束的。此时对于进程B，因为上次被换出的时候一定是在内核态且也是执行到switch\_to函数，所以即使不切换EIP，EIP的指向也是正确的，对于next task就应该指向switch\_to函数。只是内核栈变化了，执行内核代码段的上下文变化了，而且内核态的代码段是唯一的，各进程公用。

2) 此时next\_task的switch\_to函数继续执行直到完成，然后内核栈进行弹栈操作，弹出switch\_to的栈帧。同时弹出上一栈帧的EIP指针的值到EIP寄存器，恢复next\_task的运行。如下，在进行函数调用时，需要压入栈帧，压入栈帧前需要先push EIP，当弹出栈帧的时候恢复到EIP。比如A进程中是a -> b -> c -> switch\_to，此时弹出switch\_to的栈帧后，会把c的EIP恢复到eip寄存器，恢复c函数的运行。



上述的EIP指针和栈操作可以看下前文[zhuanlan.zhihu.com/p/73...](https://zhuanlan.zhihu.com/p/73...)。

switch\_to(prev, next, last): 还有一个关键点，switch\_to为什么是三个参数？而且被强制编译为寄存器传递参数。对于一次进程切换，A -> B，prev = A，next = B，但当再次切换回A时，就不一定是B了，可能是C。但是在再次切换回A时，A的内核栈prev = A，next = B，就会丢失A的前序进程 C，而context\_switch中最后一个函数finish\_task\_switch(prev)此时要求传入的prev = C，以执行一些锁的释放和硬件体系的一些回调。

此时就增加了一个last参数，是一个输出参数。

- 1) A -> B的时候，switch\_to(A, B, A)，此时prev = last。
- 2) 当C -> A的时候，switch\_to(C, A, C)，此时eax = C。当已经切换到A时，会将eax的值赋值给A内核栈中的last变量，此时prev变量的值也会变为C，这样保证A的前序进程C不丢失。


编辑于 2019-08-28

Linux

Linux 内核

计算机原理

还没有评论

写下你的评论...

想来知乎工作？请发送邮件到 [jobs@zhihu.com](mailto:jobs@zhihu.com)



