

14 | 进程数据结构（下）：项目多了就需要项目管理系统

2019-04-29 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 13:27 大小 12.33M



上两节，我们解读了 `task_struct` 的大部分的成员变量。这样一个任务执行的方方面面，都可以很好地管理起来，但是其中有一个问题我们没有谈。在程序执行过程中，一旦调用到系统调用，就需要进入内核继续执行。那如何将用户态的执行和内核态的执行串起来呢？

这就需要以下两个重要的成员变量：

复制代码

```
1 struct thread_info          thread_info;  
2 void *stack;
```

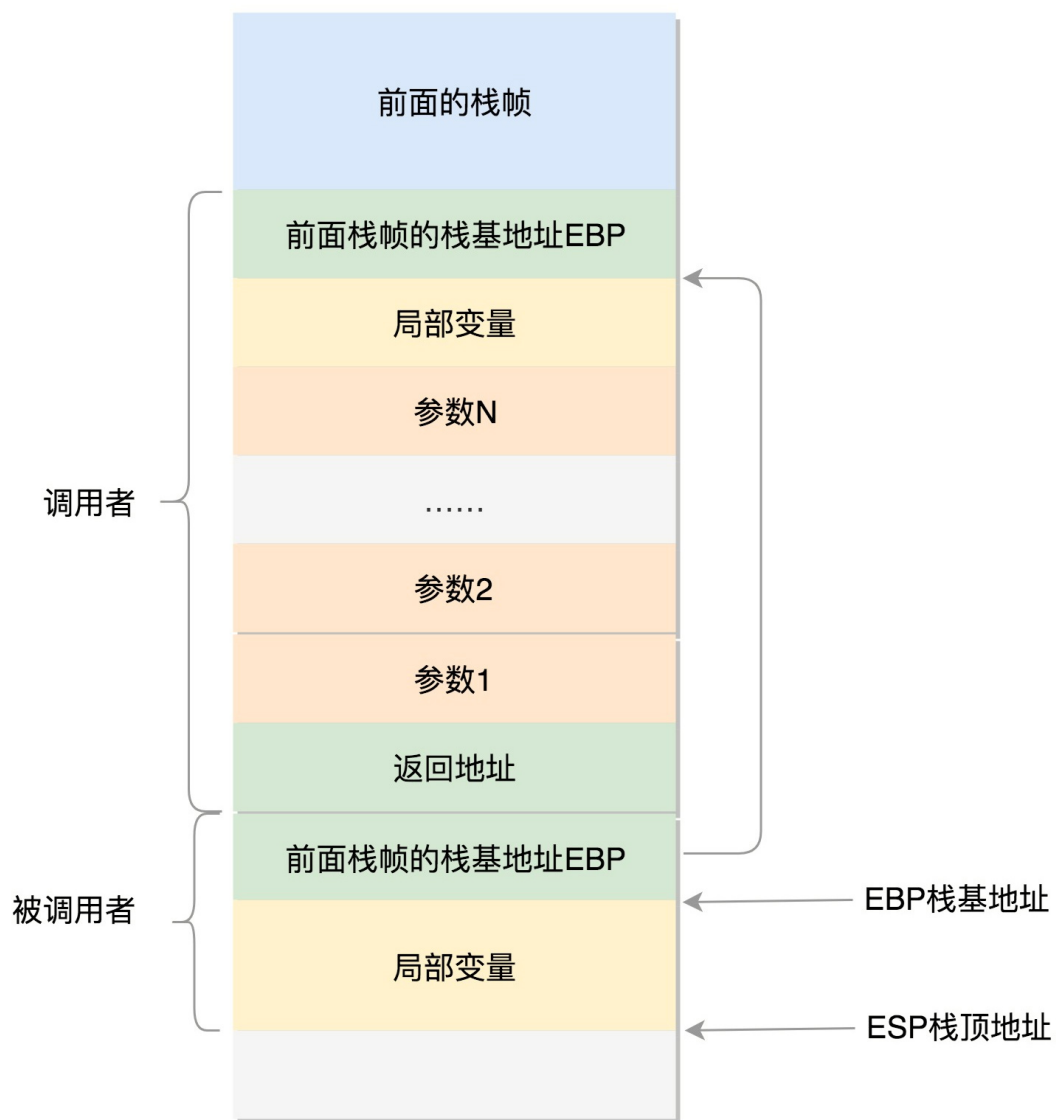
用户态函数栈

在用户态中，程序的执行往往是一个函数调用另一个函数。函数调用都是通过栈来进行的。我们前面大致讲过函数栈的原理，今天我们仔细分析一下。

函数调用其实也很简单。如果你去看汇编语言的代码，其实就是指令跳转，从代码的一个地方跳到另外一个地方。这里比较棘手的问题是，参数和返回地址应该怎么传递过去呢？

我们看函数的调用过程，A 调用 B、调用 C、调用 D，然后返回 C、返回 B、返回 A，这是一个后进先出的过程。有没有觉得这个过程很熟悉？没错，咱们数据结构里学的栈，也是后进先出的，所以用栈保存这些最合适。

在进程的内存空间里面，栈是一个从高地址到低地址，往下增长的结构，也就是上面是栈底，下面是栈顶，入栈和出栈的操作都是从下面的栈顶开始的。



我们先来看 32 位操作系统的情况。在 CPU 里，**ESP** (Extended Stack Pointer) 是栈顶指针寄存器，入栈操作 Push 和出栈操作 Pop 指令，会自动调整 ESP 的值。另外有一个寄存器**EBP** (Extended Base Pointer) ，是栈基地址指针寄存器，指向当前栈帧的最底部。

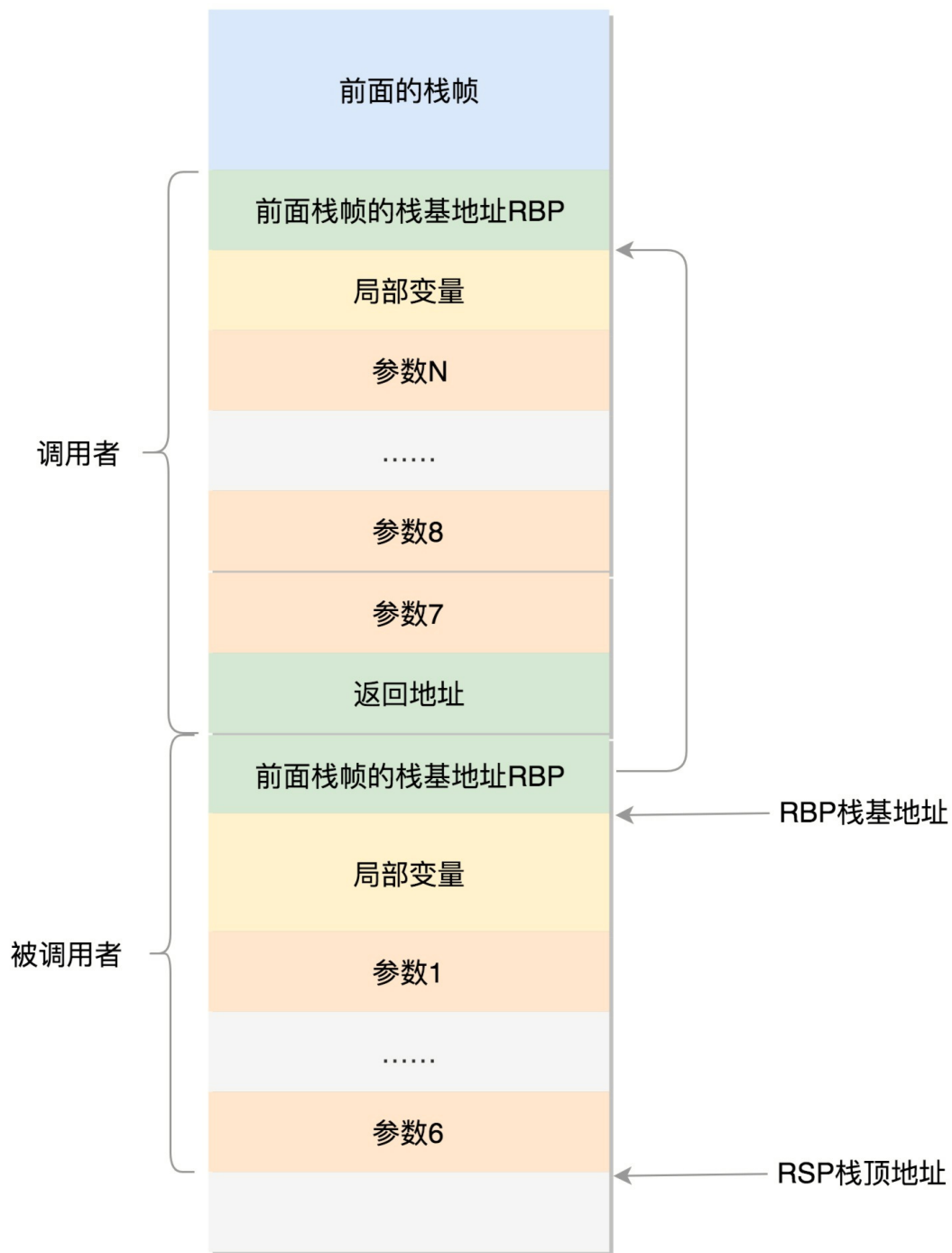
例如，A 调用 B，A 的栈里面包含 A 函数的局部变量，然后是调用 B 的时候要传给它的参数，然后返回 A 的地址，这个地址也应该入栈，这就形成了 A 的栈帧。接下来就是 B 的栈帧部分了，先保存的是 A 栈帧的栈底位置，也就是 EBP。因为在 B 函数里面获取 A 传进来的参数，就是通过这个指针获取的，接下来保存的是 B 的局部变量等等。

当 B 返回的时候，返回值会保存在 EAX 寄存器中，从栈中弹出返回地址，将指令跳转回去，参数也从栈中弹出，然后继续执行 A。

对于 64 位操作系统，模式多少有些不一样。因为 64 位操作系统的寄存器数目比较多。rax 用于保存函数调用的返回结果。栈顶指针寄存器变成了 rsp，指向栈顶位置。堆栈的 Pop 和 Push 操作会自动调整 rsp，栈基指针寄存器变成了 rbp，指向当前栈帧的起始位置。

改变比较多的是参数传递。rdi、rsi、rdx、rcx、r8、r9 这 6 个寄存器，用于传递存储函数调用时的 6 个参数。如果超过 6 的时候，还是需要放到栈里面。

然而，前 6 个参数有时候需要进行寻址，但是如果在寄存器里面，是没有地址的，因而还是会放到栈里面，只不过放到栈里面的操作是被调用函数做的。



以上的栈操作，都是在进程的内存空间里面进行的。


内核态函数栈

接下来，我们通过系统调用，从进程的内存空间到内核中了。内核中也有各种各样的函数调用来调用去的，也需要这样一个机制，这该怎么办呢？

这时候，上面的成员变量 `stack`，也就是内核栈，就派上了用场。


Linux 给每个 task 都分配了内核栈。在 32 位系统上

arch/x86/include/asm/page_32_types.h, 是这样定义的：一个 PAGE_SIZE 是 4K，左移一位就是乘以 2，也就是 8K。

 复制代码

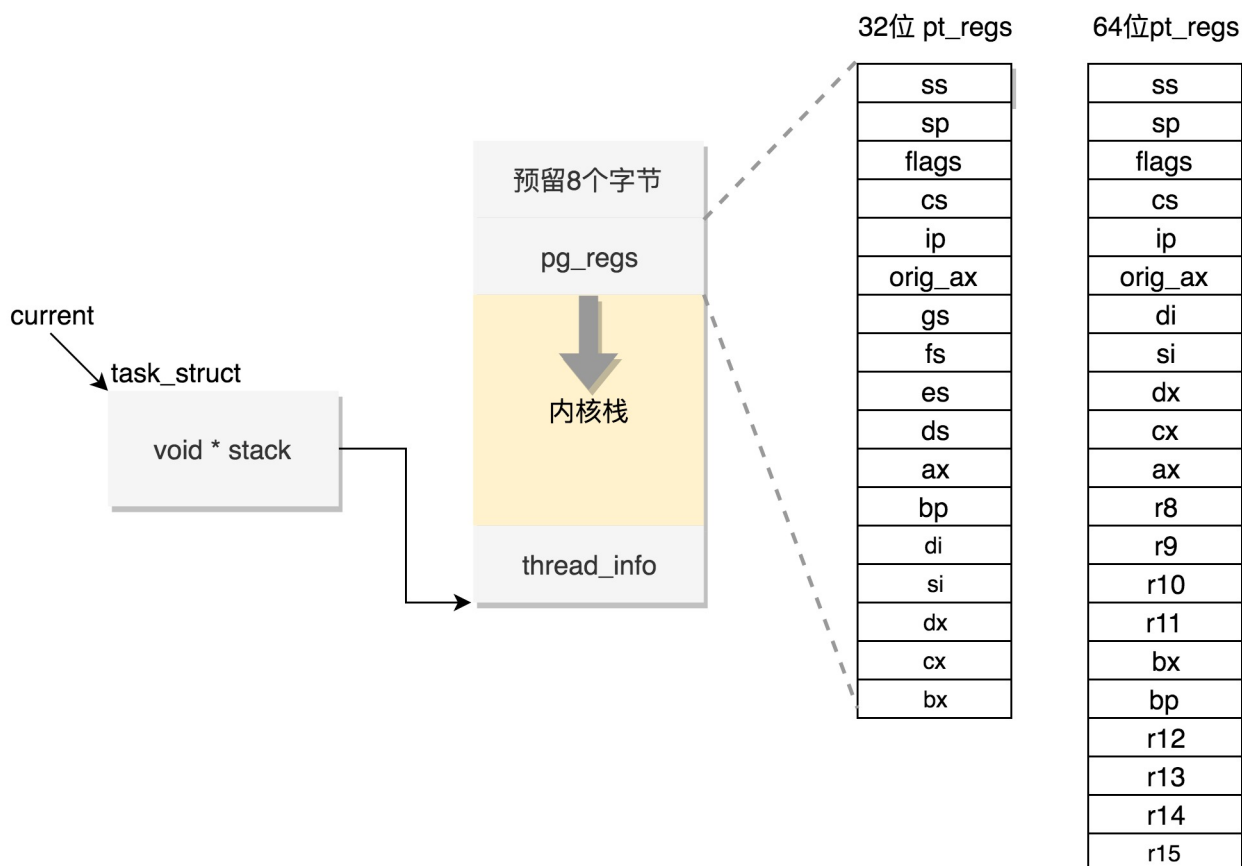
```
1 #define THREAD_SIZE_ORDER      1
2 #define THREAD_SIZE            (PAGE_SIZE << THREAD_SIZE_ORDER)
```

内核栈在 64 位系统上 arch/x86/include/asm/page_64_types.h, 是这样定义的：在 PAGE_SIZE 的基础上左移两位，也即 16K，并且要求起始地址必须是 8192 的整数倍。

 复制代码

```
1 #ifdef CONFIG_KASAN
2 #define KASAN_STACK_ORDER 1
3 #else
4 #define KASAN_STACK_ORDER 0
5 #endif
6
7
8 #define THREAD_SIZE_ORDER      (2 + KASAN_STACK_ORDER)
9 #define THREAD_SIZE            (PAGE_SIZE << THREAD_SIZE_ORDER)
```

内核栈是一个非常特殊的结构，如下图所示：



这段空间的最低位置，是一个 thread_info 结构。这个结构是对 task_struct 结构的补充。因为 task_struct 结构庞大但是通用，不同的体系结构就需要保存不同的东西，所以往往与体系结构有关的，都放在 thread_info 里面。


在内核代码里面有这样一个 union，将 thread_info 和 stack 放在一起，在 include/linux/sched.h 文件中就有。

复制代码

```
1 union thread_union {
2     #ifndef CONFIG_THREAD_INFO_IN_TASK
3         struct thread_info thread_info;
4     #endif
5     unsigned long stack[THREAD_SIZE/sizeof(long)];
6 };
```

这个 union 就是这样定义的，开头是 thread_info，后面是 stack。

在内核栈的最高地址端，存放的是另一个结构 `pt_regs`，定义如下。其中，32 位和 64 位的定义不一样。

 复制代码

```
1 #ifdef __i386__
2 struct pt_regs {
3     unsigned long bx;
4     unsigned long cx;
5     unsigned long dx;
6     unsigned long si;
7     unsigned long di;
8     unsigned long bp;
9     unsigned long ax;
10    unsigned long ds;
11    unsigned long es;
12    unsigned long fs;
13    unsigned long gs;
14    unsigned long orig_ax;
15    unsigned long ip;
16    unsigned long cs;
17    unsigned long flags;
18    unsigned long sp;
19    unsigned long ss;
20 };
21 #else
22 struct pt_regs {
23     unsigned long r15;
24     unsigned long r14;
25     unsigned long r13;
26     unsigned long r12;
27     unsigned long bp;
28     unsigned long bx;
29     unsigned long r11;
30     unsigned long r10;
31     unsigned long r9;
32     unsigned long r8;
33     unsigned long ax;
34     unsigned long cx;
35     unsigned long dx;
36     unsigned long si;
37     unsigned long di;
38     unsigned long orig_ax;
39     unsigned long ip;
40     unsigned long cs;
41     unsigned long flags;
42     unsigned long sp;
43     unsigned long ss;
44     /* top of stack page */
45 };
46 #endif
```


看到这个是不是很熟悉？咱们在讲系统调用的时候，已经多次见过这个结构。当系统调用从用户态到内核态的时候，首先要做的第一件事情，就是将用户态运行过程中的 CPU 上下文保存起来，其实主要就是保存在这个结构的寄存器变量里。这样当从内核系统调用返回的时候，才能让进程在刚才的地方接着运行下去。

如果我们对比系统调用那一节的内容，你会发现系统调用的时候，压栈的值的顺序和 `struct pt_regs` 中寄存器定义的顺序是一样的。

在内核中，CPU 的寄存器 ESP 或者 RSP，已经指向内核栈的栈顶，在内核态里的调用都有和用户态相似的过程。


通过 `task_struct` 找内核栈

如果有一个 `task_struct` 的 `stack` 指针在手，你可以通过下面的函数找到这个线程内核栈：

 复制代码

```
1 static inline void *task_stack_page(const struct task_struct *task)
2 {
3     return task->stack;
4 }
```

从 `task_struct` 如何得到相应的 `pt_regs` 呢？我们可以通过下面的函数：

 复制代码


```
1 /*
2  * TOP_OF_KERNEL_STACK_PADDING reserves 8 bytes on top of the ring0 stack.
3  * This is necessary to guarantee that the entire "struct pt_regs"
4  * is accessible even if the CPU haven't stored the SS/ESP registers
5  * on the stack (interrupt gate does not save these registers
6  * when switching to the same priv ring).
7  * Therefore beware: accessing the ss/esp fields of the
8  * "struct pt_regs" is possible, but they may contain the
9  * completely wrong values.
10 */
11 #define task_pt_regs(task) \
12 ({ \
13     unsigned long __ptr = (unsigned long)task_stack_page(task); \
```



```
14     __ptr += THREAD_SIZE - TOP_OF_KERNEL_STACK_PADDING;           \
15     ((struct pt_regs *)__ptr) - 1;                                   \
16 })
```

你会发现，这是先从 `task_struct` 找到内核栈的开始位置。然后这个位置加上 `THREAD_SIZE` 就到了最后的位置，然后转换为 `struct pt_regs`，再减一，就相当于减少了一个 `pt_regs` 的位置，就到了这个结构的首地址。

这里面有一个 `TOP_OF_KERNEL_STACK_PADDING`，这个的定义如下：

 复制代码

```
1 #ifdef CONFIG_X86_32
2 # ifdef CONFIG_VM86
3 #   define TOP_OF_KERNEL_STACK_PADDING 16
4 # else
5 #   define TOP_OF_KERNEL_STACK_PADDING 8
6 # endif
7 #else
8 # define TOP_OF_KERNEL_STACK_PADDING 0
9 #endif
```

也就是说，32 位机器上是 8，其他是 0。这是为什么呢？因为压栈 `pt_regs` 有两种情况。我们知道，CPU 用 ring 来区分权限，从而 Linux 可以区分内核态和用户态。

因此，第一种情况，我们拿涉及从用户态到内核态的变化的系统调用来说。因为涉及权限的改变，会压栈保存 `SS`、`ESP` 寄存器的，这两个寄存器共占用 8 个 byte。


另一种情况是，不涉及权限的变化，就不会压栈这 8 个 byte。这样就会使得两种情况不兼容。如果没有压栈还访问，就会报错，所以还不如预留在这里，保证安全。在 64 位上，修改了这个问题，变成了定长的。

好了，现在如果你 `task_struct` 在手，就能够轻松得到内核栈和内核寄存器。

通过内核栈找 `task_struct`

那如果一个当前在某个 CPU 上执行的进程，想知道自己的 task_struct 在哪里，又该怎么办呢？

这个艰巨的任务要交给 thread_info 这个结构。

 复制代码

```
1 struct thread_info {
2     struct task_struct    *task;          /* main task structure */
3     __u32                 flags;          /* low level flags */
4     __u32                 status;         /* thread synchronous flags */
5     __u32                 cpu;           /* current CPU */
6     mm_segment_t          addr_limit;
7     unsigned int          sig_on_uaccess_error:1;
8     unsigned int          uaccess_err:1; /* uaccess failed */
9 };
```


这里面有个成员变量 task 指向 task_struct，所以我们常用 current_thread_info()->task 来获取 task_struct。

 复制代码

```
1 static inline struct thread_info *current_thread_info(void)
2 {
3     return (struct thread_info *)(current_top_of_stack() - THREAD_SIZE);
4 }
```

而 thread_info 的位置就是内核栈的最高位置，减去 THREAD_SIZE，就到了 thread_info 的起始地址。


但是现在变成这样了，只剩下一个 flags。

 复制代码

```
1 struct thread_info {
2     unsigned long         flags;          /* low level flags */
3 };
```


那这时候怎么获取当前运行中的 task_struct 呢？current_thread_info 有了新的实现方式。

在 include/linux/thread_info.h 中定义了 current_thread_info。

 复制代码

```
1 #include <asm/current.h>
2 #define current_thread_info() ((struct thread_info *)current)
3 #endif
```

那 current 又是什么呢？在 arch/x86/include/asm/current.h 中定义了。

 复制代码


```
1 struct task_struct;
2
3
4 DECLARE_PER_CPU(struct task_struct *, current_task);
5
6
7 static __always_inline struct task_struct *get_current(void)
8 {
9     return this_cpu_read_stable(current_task);
10 }
11
12
13 #define current get_current
```

到这里，你会发现，新的机制里面，每个 CPU 运行的 task_struct 不通过 thread_info 获取了，而是直接放在 Per CPU 变量里面了。

多核情况下，CPU 是同时运行的，但是它们共同使用其他的硬件资源的时候，我们需要解决多个 CPU 之间的同步问题。

Per CPU 变量是内核中一种重要的同步机制。顾名思义，Per CPU 变量就是为每个 CPU 构造一个变量的副本，这样多个 CPU 各自操作自己的副本，互不干涉。比如，当前进程的变量 current_task 就被声明为 Per CPU 变量。

要使用 Per CPU 变量，首先要声明这个变量，在 arch/x86/include/asm/current.h 中有：

 复制代码

```
1 DECLARE_PER_CPU(struct task_struct *, current_task);
```

然后是定义这个变量，在 arch/x86/kernel/cpu/common.c 中有：

 复制代码

```
1 DEFINE_PER_CPU(struct task_struct *, current_task) = &init_task;
```

也就是说，系统刚刚初始化的时候，current_task 都指向 init_task。

当某个 CPU 上的进程进行切换的时候，current_task 被修改为将要切换到的目标进程。例如，进程切换函数 __switch_to 就会改变 current_task。

 复制代码

```
1 __visible __notrace_funcgraph struct task_struct *
2 __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
3 {
4     .....
5     this_cpu_write(current_task, next_p);
6     .....
7     return prev_p;
8 }
```

当要获取当前的运行中的 task_struct 的时候，就需要调用 this_cpu_read_stable 进行读取。

 复制代码

```
1 #define this_cpu_read_stable(var)      percpu_stable_op("mov", var)
```

好了，现在如果你是一个进程，正在某个 CPU 上运行，就能够轻松得到 task_struct 了。

总结时刻

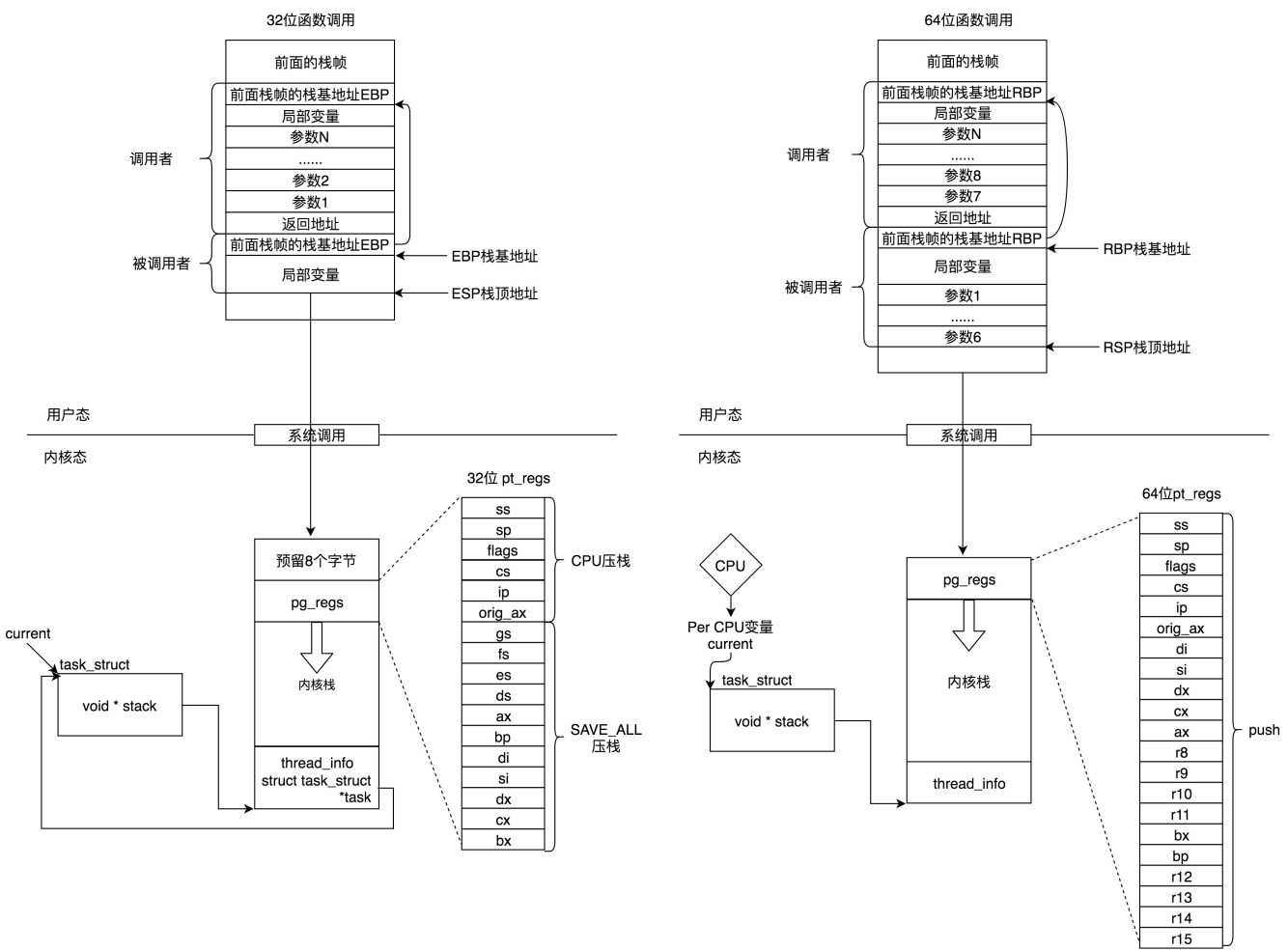
这一节虽然只介绍了内核栈，但是内容更加重要。如果说 task_struct 的其他成员变量都是和进程管理有关的，内核栈是和进程运行有关系的。

我这里画了一张图总结一下 32 位和 64 位的工作模式，左边是 32 位的，右边是 64 位的。

在用户态，应用程序进行了至少一次函数调用。32 位和 64 位的传递参数的方式稍有不同，32 位的就是用函数栈，64 位的前 6 个参数用寄存器，其他的用函数栈。

在内核态，32 位和 64 位都使用内核栈，格式也稍有不同，主要集中在 pt_regs 结构上。

在内核态，32 位和 64 位的内核栈和 task_struct 的关联关系不同。32 位主要靠 thread_info，64 位主要靠 Per-CPU 变量。



课堂练习

这一节讲函数调用的时候，我们讲了函数栈的工作模式。请你写一个程序，然后编译为汇编语言，打开看一下，函数栈是如何起作用的。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，**反复研读**。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

 极客时间

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 学习攻略（一）：学好操作系统，需要掌握哪些前置知识？

下一篇 15 | 调度（上）：如何制定项目管理流程？

精选留言 (12)

写留言



why

2019-04-30

14

- 用户态/内核态切换执行如何串起来
- 用户态函数栈; 通过 JMP + 参数 + 返回地址 调用函数

- 栈内存空间从高到低增长
- 32位栈结构: 栈帧包含 前一个帧的 EBP + 局部变量 + N个参数 + 返回地址
 - ESP: 栈顶指针; EBP: 栈基址(栈帧最底部, 局部变量起始)...

展开 ▾



第九天魔王

2019-04-29

👍 2

从网络协议课学到这里，满满的干货啊。特别是配图，太用心了，一看就是非常有心才能做出来的。谢谢刘老师！

作者回复: 谢谢鼓励

◀ ▶



青石

2019-04-29

👍 1

在12节里面提到“Linux的任务管理由统一的结构task_struct进行管理”。那么多核CPU的任务切换时，是不是就是将current_task切换到另一个task_struct呢？

THREAD_SIZE是固定的大小，32位系统中是8K（页大小左移一位），64位系统是16K（页大小左移二位）。TOP_OF_KERNEL_STACK_PADDING就是图“内核栈是一个...

展开 ▾

作者回复: 不仅仅切换这个变量，要切换的还挺多的

◀ ▶



青石

2019-04-29

👍 1

看这篇内容时，查了几篇资料。

在汇编代码中，函数调用的参数传递是通过把参数依次放在靠近调用者的栈的顶部来实现的。调用者获取参数时，只要相对于当前帧指针的向上偏移即可取到参数。即取调用者函数参数时执行movl 8(%ebp), %edx。

展开 ▾



hua168

👍 1



2019-04-29

老师，我求您了，能不能把您讲的专栏中涉及的书名加个豆瓣之类的链接🙏🙏，有些都重名，作者不同，🙏🙏不带这样玩的🙏🙏🙏

作者回复: 啊，好的，看来应该给英文名加作者的



石维康

2019-04-29

👍 1

文中说“接下来就是 B 的栈帧部分了，先保存的是 A 栈帧的栈底位置,也就是 EBP。因为在 B 函数里面获取 A 传进来的参数，就是通过这个指针获取的，”感觉主流编译器还是直接能通过当前 RBP 或者 RSP 来进行偏移定位到传进来的参数了吧？保存这个 A 栈底位置更多的是为了回复 A 的现场吧？



Liber

2019-04-29

👍 1

0点自动发，应该是自动发布的吧

展开 ∨



嘉木

2019-05-28

👍

有个疑问，两个进程切换时，用户栈的上下文保存在哪？

展开 ∨

作者回复: 后面讲切换的时候会讲，用户栈相关寄存器在pt_regs中，用户栈的内存存在虚拟地址空间，不会冲突



骨汤鸡蛋面

2019-05-22

👍

为什么用户态的栈不需要 task_struct 维护一个类似 *stack 的指针呢？

展开 ∨





周平
2019-05-09



虽然还有些迷糊，但是感觉自己的理解在加深了

展开 ▾



川云
2019-05-05



讲的真好，把之前用户态切内核态不明白的地方都捋清了

展开 ▾



安排
2019-04-30



内核栈的最高地址的那8个字节没看懂，如果没压入ss,sp，那为什么还要访问呢？从源码注释看，加了8个字节保留，如果没有压入ss,sp,这时候访问也是没有意义的，既然没有意义，程序为什么还要访问他啊？一个在程序中不能控制吗？

作者回复: 不会访问，但是访问到是错的，也比越界好。这样统一处理会方便很多。当然一路if else也是可以的

