Linux 中的各种栈

栈有什么作用?

一、函数调用

我们知道一个函数调用有以下三个基本过程:

- 调用参数的传入
- 局部变量的空间管理
- 函数返回

函数的调用必须是高效的，而数据存放在 CPU通用寄存器 或者 RAM 内存 中无疑是最好的选择。以传递调用参数为例，我们可以选择使用 CPU通用寄存器 来存放参数。但是通用寄存器的数目都是有限的，当出现函数嵌套调用时，子函数再次使用原有的通用寄存器必然会导致冲突。因此如果想用它来传递参数，那在调用子函数前，就必须先 保存原有寄存器的值，然后当子函数退出的时候再 恢复原有寄存器的值 。

函数的调用参数数目一般都相对少，因此通用寄存器是可以满足一定需求的。但是局部变量的数目和占用空间都是比较大的，再依赖有限的通用寄存器未免强人所难，因此我们可以采用某些 RAM 内存区域来存储局部变量。但是存储在哪里合适？既不能让函数嵌套调用的时候有冲突，又要注重效率。

这种情况下，栈无疑提供很好的解决办法。一、对于通用寄存器传参的冲突，我们可以再调用子函数前，将通用寄存器临时压入栈中；在子函数调用完毕后，在将已保存的寄存器再弹出恢复回来。二、而局部变量的空间申请，也只需要向下移动下栈顶指针；将栈顶指针向回移动，即可就可完成局部变量的空间释放；三、对于函数的返回，也只需要在调用子函数前，将返回地址压入栈中，待子函数调用结束后，将函数返回地址弹出给 PC 指针，即完成了函数调用的返回;

于是上述函数调用的三个基本过程，就演变记录一个栈指针的过程。每次函数调用的时候，都配套一个栈指针。即使循环嵌套调用函数，只要对应函数栈指针是不同的，也不会出现冲突。

函数调用时入栈的顺序为:
实参N~1 → 主调函数返回地址 → 主调函数帧基指针EBP → 被调函数局部变量1~N

二、多任务支持

然而栈的意义还不只是函数调用，有了它的存在，才能构建出操作系统的多任务模式。我们以 main 函数调用为例，main 函数包含一个无限循环体，循环体中先调用 A 函数，再调用 B 函数。

```
1 func B():
2   return;
3
```

```
4  func A():
5    B();
6
7  func main():
8    while (1)
9      A();
```

试想在单处理器情况下，程序将永远停留在此 main 函数中。即使有另外一个任务在等待状态，程序是没法从此 main 函数里面跳转到另一个任务。因为如果是函数调用关系，本质上还是属于 main 函数的任务中，不能算多任务切换。此刻的 main 函数任务本身其实和它的栈绑定在了一起，无论如何嵌套调用函数，栈指针都在本栈范围内移动。

由此可以看出一个任务可以利用以下信息来表征：

1. main 函数体代码
2. main 函数栈指针
3. 当前 CPU 寄存器信息

假如我们可以保存以上信息，则完全可以强制让出 CPU 去处理其他任务。只要将来想继续执行此 main 任务的时候，把上面的信息恢复回去即可。有了这样的先决条件，多任务就有了存在的基础，也可以看出栈存在的另一个意义。在多任务模式下，当调度程序认为有必要进行任务切换的话，只需保存任务的信息（即上面说的三个内容）。恢复另一个任务的状态，然后跳转到上次运行的位置，就可以恢复运行了。

可见每个任务都有自己的栈空间，正是有了独立的栈空间，为了代码重用，不同的任务甚至可以混用任务的函数体本身，例如可以一个main函数有两个任务实例。至此之后的操作系统的框架也形成了，譬如任务在调用 sleep() 等待的时候，可以主动让出 CPU 给别的任务使用，或者分时操作系统任务在时间片用完是也会被迫的让出 CPU。不论是哪种方法，只要想办法切换任务的上下文空间，切换栈即可。

一、进程栈

进程虚拟地址空间中的栈区，正指的是我们所说的进程栈。进程栈的初始化大小是由编译器和链接器计算出来的，但是栈的实时大小并不是固定的，Linux 内核会根据入栈情况对栈区进行动态增长（其实也就是添加新的页表）。但是并不是说栈区可以无限增长，它也有最大限制 RLIMIT_STACK (一般为 8M)，我们可以通过 ulimit 来查看或更改 RLIMIT_STACK 的值。

二、线程栈

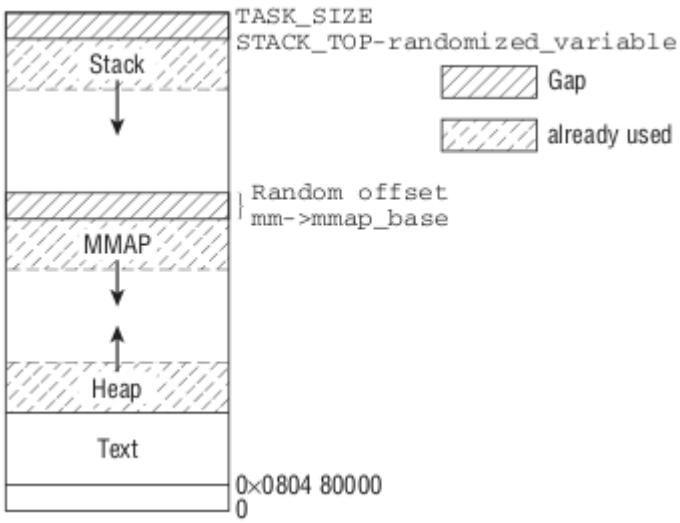
从 Linux 内核的角度来说，其实它并没有线程的概念。Linux 把所有线程都当做进程来实现，它将线程和进程不加区分的统一到了 `task_struct` 中。线程仅仅被视为一个与其他进程共享某些资源的进程，而是否共享地址空间几乎是进程和 Linux 中所谓线程的唯一区别。线程创建的时候，加上了 `CLONE_VM` 标记，这样 线程的内存描述符 将直接指向 父进程的内存描述符。

```
1   if (clone_flags & CLONE_VM) {
2     /*
3      * current 是父进程而 tsk 在 fork() 执行期间是共享子进程
4      */
5     atomic_inc(&current->mm->mm_users);
6     tsk->mm = current->mm;
7   }
```

虽然线程的地址空间和进程一样，但是对待其地址空间的 stack 还是有些区别的。对于 Linux 进程或者说主线程，其 stack 是在 fork 的时候生成的，实际上就是复制了父亲的 stack 空间地址，然后写时拷贝 (cow) 以及动态增长。然而对于主线程生成的子线程而言，其 stack 将不再是这样的了，而是事先固定下来的，使用 mmap 系统调用，它不带有 VM_STACK_FLAGS 标记。这个可以从 glibc 的 `nptl/allocatestack.c` 中的 `allocate_stack()` 函数中看到：

```
1  mem = mmap (NULL, size, prot, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0);
```

这也就意味着线程栈所在的位置是在mmap区。



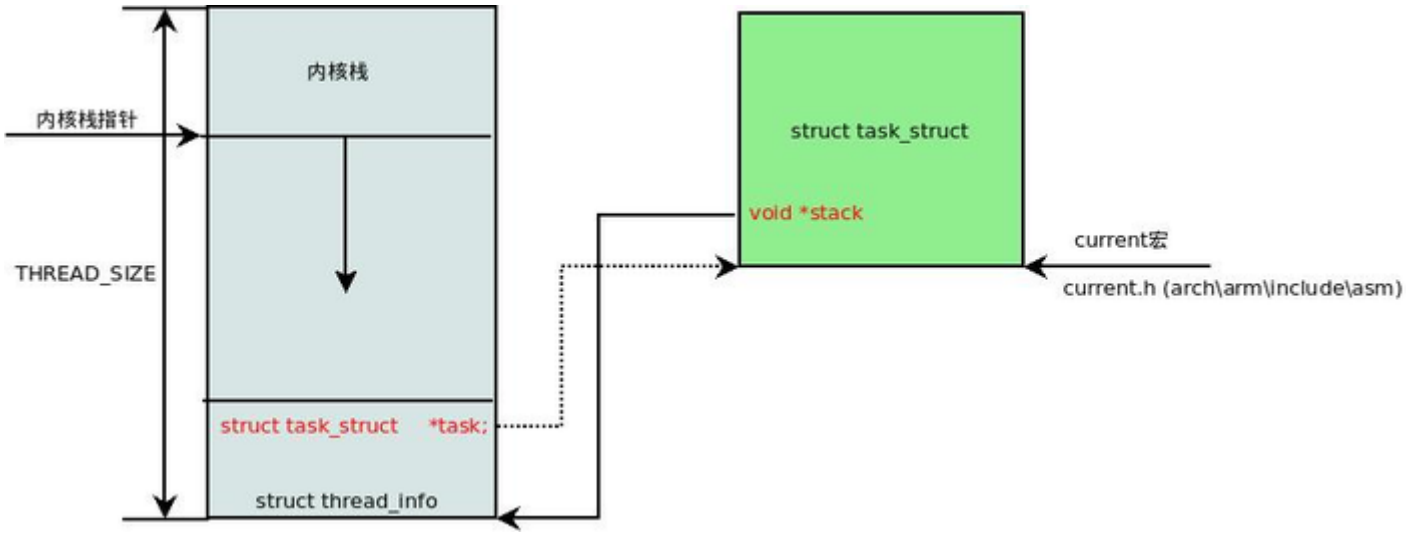
由于线程的 mm->start_stack 栈地址和所属进程相同，所以线程栈的起始地址并没有存放在 `task_struct` 中，应该是使用 `pthread_attr_t` 中的 `stackaddr` 来初始化 `task_struct->thread->sp` （sp 指向 `struct pt_regs` 对象，该结构体用于保存用户进程或者线程的寄存器现场）。这些都不重要，重要的是，线程栈不能动态增长，一旦用尽就没了，这是和生成进程的 fork 不同的地方。由于线程栈是从进程的地址空间中 map 出来的一块内存区域，原则上是线程私有的。但是同一个进程的所有线程生成的时候浅拷贝生成者的 `task_struct` 的很多字段，其中包括所有的 vma，如果愿意，其它线程也还是可以访问到的，于是一定要注意。

三、进程内核栈

在每一个进程的生命周期中，必然会通过到系统调用陷入内核。在执行系统调用陷入内核之后，这些内核代码所使用的栈并不是原先进程用户空间中的栈，而是一个单独内核空间的栈，这个称作进程内核栈。进程内核栈在进程创建的时候，通过 slab 分配器从 `thread_info_cache` 缓存池中分配出来，其大小为 `THREAD_SIZE`，一般来说是一个页大小 4K；

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

`thread_union` 进程内核栈 和 `task_struct` 进程描述符有着紧密的联系。由于内核经常要访问 `task_struct`，高效获取当前进程的描述符是一件非常重要的事情。因此内核将进程内核栈的头部一段空间，用于存放 `thread_info` 结构体，而此结构体中则记录了对应进程的描述符，两者关系如下图（对应内核函数为 `dup_task_struct()`）：



有了上述关联结构后，内核可以先获取到栈顶指针 esp，然后通过 esp 来获取 thread_info。这里有一个小技巧，直接将 esp 的地址与上 ~(THREAD_SIZE - 1) 后即可直接获得 thread_info 的地址。由于 thread_union 结构体是从 thread_info_cache 的 Slab 缓存池中申请出来的，而 thread_info_cache 在 kmem_cache_create 创建的时候，保证了地址是 THREAD_SIZE 对齐的。因此只需要对栈指针进行 THREAD_SIZE 对齐，即可获得 thread_union 的地址，也就获得了 thread_union 的地址。成功获取到 thread_info 后，直接取出它的 task 成员就成功得到了 task_struct。其实上面这段描述，也就是 current 宏的实现方法：

```
register unsigned long current_stack_pointer asm ("sp");
static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)
                (current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

```
7
8  #define get_current() (current_thread_info()->task)
9
10 #define current get_current()
```

四、中断栈

进程陷入内核态的时候，需要内核栈来支持内核函数调用。中断也是如此，当系统收到中断事件后，进行中断处理的时候，也需要中断栈来支持函数调用。由于系统中断的时候，系统当然是处于内核态的，所以中断栈是可以和内核栈共享的。但是具体是否共享，这和具体处理架构密切相关。

X86 上中断栈就是独立于内核栈的；独立的中断栈所在内存空间的分配发生在 `arch/x86/kernel/irq_32.c` 的 `irq_ctx_init()` 函数中 (如果是多处理器系统，那么每个处理器都会有一个独立的中断栈)，函数使用 `__alloc_pages` 在低端内存区分配 2个物理页面，也就是8KB大小的空间。有趣的是，这个函数还会为 softirq 分配一个同样大小的独立堆栈。如此说来，softirq 将不会在 hardirq 的中断栈上执行，而是在自己的上下文中执行。

而 ARM 上中断栈和内核栈则是共享的；中断栈和内核栈共享有一个负面因素，如果中断发生嵌套，可能会造成栈溢出，从而可能会破坏到内核栈的一些重要数据，所以栈空间有时候难免会捉襟见肘。