Linux线程之线程栈

我们接上一篇继续学习，这一篇的重点放在线程栈上。

我们用过 pthread_create 接口，也用过 pthread_self 接口，请看 manual 中的声明：

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
pthread_t pthread_self(void)
```

我们看到，pthread_create 的第一个参数是 pthread_t 类型的指针，函数会将一个值填入该指针对应的内存？那么这个值是什么？pthread_self 会返回一个 pthread_t 类型的值，它又是什么？传说中的 GS 寄存器和线程栈有神马关系？请看测试代码，还是相同的测试代码：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <assert.h>

#define gettid() syscall(__NR_gettid)

pthread_key_t key;
__thread int count = 2222;
__thread unsigned long long count2 ;
static __thread int count3;
void echomsg(char* string)
{
     printf("destructor excuted in thread %x,address (%p) param%s\n",
             pthread_self(),string,string);
     free(string);
}

void * child1(void *arg)
{
     int b;
     int tid=pthread_self();

     printf("I am the child1 pthread_self return %p gettid return %d\n",tid,gettid());

     char* key_content = malloc(8);
     if(key_content != NULL)
     {
          strcpy(key_content,"ACACACA");
     }
     pthread_setspecific(key,(void *)key_content);

     count=666666;
     count2=1023;
```

```
    count3=2048;
    printf("I am child1 , tid=%x ,count (%p) = %8d,count2(%p) = %6llu,count3(%p) =
%6d\n",tid,&count,count,&count2,count2,&count3,count3);
    asm volatile("movl %%gs:0, %0;"
             :"=r"(b) /* output */
             );

    printf("I am child1 , GS address %p\n",b);

    sleep(2);
    printf("thread %x returns %x\n",tid,pthread_getspecific(key));
    sleep(50);
}

void * child2(void *arg)
{
    int b;
    int tid=pthread_self();

    printf("I am the child2 pthread_self return %p gettid return %d\n",tid,gettid());

    char* key_content = malloc(8);
    if(key_content != NULL)
    {
        strcpy(key_content,"ABCDEFG");
    }
    pthread_setspecific(key,(void *)key_content);
    count=88888888;
    count2=1024;
    count3=2047;
    printf("I am child2 , tid=%x ,count (%p) = %8d,count2(%p) = %6llu,count3(%p) =
%6d\n",tid,&count,count,&count2,count2,&count3,count3);


    asm volatile("movl %%gs:0, %0;"
             :"=r"(b) /* output */
             );

    printf("I am child2 , GS address %p\n",b);

    sleep(1);
    printf("thread %x returns %x\n",tid,pthread_getspecific(key));
    sleep(50);
}
```

```
int main(void)
{
     int b;
     pthread_t tid1,tid2;
     printf("hello\n");


     pthread_key_create(&key,echomsg);

     asm volatile("movl %%gs:0, %0;"
               :"=r"(b) /* output */
               );

     printf("I am the main , GS address %x\n",b);

     pthread_create(&tid1,NULL,child1,NULL);
     pthread_create(&tid2,NULL,child2,NULL);

     printf("pthread_create tid1 = %p\n",tid1);
     printf("pthread_create tid2 = %p\n",tid2);

     sleep(60);
     pthread_key_delete(key);
     printf("main thread exit\n");
     return 0;
}
```

中间嵌入了一段汇编代码，代码的本意是取出 GS 指示的段（对 GS 不了解的可以查看这篇博文，touch me）。

```
root@manu:~/code/c/self/tls# ./test2
hello
I am the main , GS address b7531700
pthread_create tid1 = 0xb7530b40
pthread_create tid2 = 0xb6d2fb40
I am the child2  pthread_self return 0xb6d2fb40 gettid return 10061
I am the child1  pthread_self return 0xb7530b40 gettid return 10060
I am child2 , tid=b6d2fb40 ,count (0xb6d2fb28) = 88888888,count2(0xb6d2fb30) =   1024,count3(0xb6d2fb38) =   2047
I am child2 , GS address 0xb6d2fb40
I am child1 , tid=b7530b40 ,count (0xb7530b28) =   666666,count2(0xb7530b30) =   1023,count3(0xb7530b38) =   2048
I am child1 , GS address 0xb7530b40
thread b6d2fb40 returns 0xb6400468
thread b7530b40 returns 0xb6200468
destructor excuted in thread b6d2fb40,address (0xb6400468) param=ABCDEFG
destructor excuted in thread b7530b40,address (0xb6200468) param=ACACACA
main thread exit
root@manu:~/code/c/self/tls#
```

我们惊奇的发现对于 child1

1 pthread_create 第一参数返回 pthread_t 类型的值为 0xb7530b40

2 pthread_self 返回的 pthread_t 类型的值为 0xb7530b40

3 GS 指示的段（GDT 的第六个段）存储的内容还是 0xb7530b40

对于 child2 也有类似的情况，三者返回同一个值，what is the magic number mean？只能求助 glibc。幸好我们有了源码。首先从 pthread_create 搞起。

代码在 nptl 目录下的 pthread_create.c 下面，比较有意思的是居然没有一个函数叫 pthread_create。

```
__pthread_create_2_0
__pthread_create_2_1
compat_symbol (libpthread, __pthread_create_2_0, pthread_create, GLIBC_2_0)
```

七度黑光对这个问题有一篇专门的博文解释（touch me），我就不纠缠细节了，总之，pthread_create_2_0 调用了 pthread_create_2_1，而后者才是真正干活的函数，参数都一样：

```
int
__pthread_create_2_1 (newthread, attr, start_routine, arg)
     pthread_t *newthread;
     const pthread_attr_t *attr;
     void *(*start_routine) (void *);
     void *arg
{
      ...
     struct pthread *pd = NULL;
     int err = ALLOCATE_STACK (iattr, &pd);
      ....
     /* Pass the descriptor to the caller. */
     *newthread = (pthread_t) pd;

     /* Start the thread. */
     return create_thread (pd, iattr, STACK_VARIABLES_ARGS);

}
```

看到了，newthread 就是我们传入的地址，它在最后被赋值为 pd，pd 是在 ALLOTCATE_STACK 里面赋的值。

ALLOCATE_STACK，我的智商不高，我也看出来它老人家用处是给线程分配栈的。比较下图，ALLOCATE_STACK 之前和之后，虚拟地址空间变化。最主要的变化是多了 8200KB 的一块内存空间。这块区域是在 allocate_stack（ALLOCATE_STACK 是个宏，本质是 allocate_stack 函数）函数里面分配的。

```
root@manu:~/code/c/self/hello# pmap 7384
7384:   /home/manu/code/c/self/tls/test2
08048000      4K r-x--  /home/manu/code/c/self/tls/test2
08049000      4K r----  /home/manu/code/c/self/tls/test2
0804a000      4K rw---  /home/manu/code/c/self/tls/test2
b7dfa000      8K rw---    [ anon ]
b7dfc000   1676K r-x--  /lib/i386-linux-gnu/libc-2.15.so
b7f9f000      8K r----  /lib/i386-linux-gnu/libc-2.15.so
b7fa1000      4K rw---  /lib/i386-linux-gnu/libc-2.15.so
b7fa2000     12K rw---    [ anon ]
b7fa5000     92K r-x--  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbc000      4K r----  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbd000      4K rw---  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbe000      8K rw---    [ anon ]
b7fda000     12K rw---    [ anon ]
b7fdd000      4K r-x--    [ anon ]
b7fde000    128K r-x--  /lib/i386-linux-gnu/ld-2.15.so
b7ffe000      4K r----  /lib/i386-linux-gnu/ld-2.15.so
b7fff000      4K rw---  /lib/i386-linux-gnu/ld-2.15.so
bffdf000    132K rw---    [ stack ]
 total     2112K
```

```
root@manu:~/code/c/self/hello# pmap 7384
7384:   /home/manu/code/c/self/tls/test2
08048000      4K r-x--  /home/manu/code/c/self/tls/test2
08049000      4K r----  /home/manu/code/c/self/tls/test2
0804a000      4K rw---  /home/manu/code/c/self/tls/test2
0804b000    132K rw---    [ anon ]
b75f9000      4K -----    [ anon ]
b75fa000   8200K rw---    [ anon ]
b7dfc000   1676K r-x--  /lib/i386-linux-gnu/libc-2.15.so
b7f9f000      8K r----  /lib/i386-linux-gnu/libc-2.15.so
b7fa1000      4K rw---  /lib/i386-linux-gnu/libc-2.15.so
b7fa2000     12K rw---    [ anon ]
b7fa5000     92K r-x--  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbc000      4K r----  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbd000      4K rw---  /lib/i386-linux-gnu/libpthread-2.15.so
b7fbe000      8K rw---    [ anon ]
b7fda000     12K rw---    [ anon ]
b7fdd000      4K r-x--    [ anon ]
b7fde000    128K r-x--  /lib/i386-linux-gnu/ld-2.15.so
b7ffe000      4K r----  /lib/i386-linux-gnu/ld-2.15.so
b7fff000      4K rw---  /lib/i386-linux-gnu/ld-2.15.so
bffdf000    132K rw---    [ stack ]
 total    10440K
```

在分析这个 `allocate_stack` 之前，需要指出的一点是还没有调用 `clone` 系统调用，也就是还没到 `kernel` 呢，更没有分配 `task_struct` 等等。好，开始分析：

```
struct pthread *pd;
 size_t size;
 size_t pagesize_m1 = __getpagesize () - 1;
 void *stacktop;

 assert (attr != NULL);
 assert (powerof2 (pagesize_m1 + 1));
 assert (TCB_ALIGNMENT >= STACK_ALIGN);
```

```
    /* Get the stack size from the attribute if it is set. Otherwise we
       use the default we determined at start time. */
    size = attr->stacksize ?: __default_stacksize;//此处决定了size是8M，如果user指定了stack_size此处会是用户指定的值。

    /* Get memory for the stack. */
    if (__builtin_expect (attr->flags & ATTR_FLAG_STACKADDR, 0))
    {
        ...
    }
    else
    {
        ...
    }
```

加粗的一行含义是，如果用户指定了stacksize，用attr里面的指定值，否则，默认值。至于__default_stacksize可以通过ulimit -s查看。一般是8192KB。

至于代码中的if/else，如果用户指定了stack的基址（pthread_attr_setstack）走入if分支，否则走入else分支，我们是普通青年，轻易不会干pthread_attr_setstack这么妖娆的事情，所以我们走入else分支。

```
        pd = get_cached_stack (&size, &mem);
        if (pd == NULL)
        {
            /* To avoid aliasing effects on a larger scale than pages we
               adjust the allocated stack size if necessary. This way
               allocations directly following each other will not have
               aliasing problems. */
#if MULTI_PAGE_ALIASING != 0
            if ((size % MULTI_PAGE_ALIASING) == 0)
                size += pagesize_m1 + 1;
#endif

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

            if (__builtin_expect (mem == MAP_FAILED, 0))
                return errno
```

在尝试mmap分配之前，会首先使用get_cached_stack查找下，有没有现成可用的堆栈空间。说的这，有些筒子可能迷惑，啥叫现成可用的呢？我们创建了一个线程，然后线程完成了他的使命，线程退出了，但是线程退出并不意味这线程的堆栈就要释放。如果A线程退出后，我们又需要创建一个新线程B，那么我们就可以看看A线程的堆栈空间是否满足要求，满足要求的话我们就直接用了。这就是get_cached_stack的含义。

这个例子告诉我们，线程退出之后，它占据的堆栈空间还在，如果这种属性不是我们期望的，NPTL提供了两个方法：首当其冲的是 pthread_join。简单说叫起线程的这个主 LWP 可以调用 pthread_join 为线程收尸，销毁线程的资源。主 LWP 用 pthread_create 创建了线程，然后 pthread_join 为退出的线程销毁资源，有种白发人送黑发人的感觉。这种方法不好的地方在于阻塞，主 LWP 会堵在此处，直到线程推出。那第二个方法就是 pthread_detach(pthread_self()),意思线程自己会把后事交代清楚，线程退出前，自会自我了断，该释放的资源都会释放。

我们是初次创建线程，get_cached_stack 自然是无功而返。但是 MULTI_PAGE_ALIASING=64KB，我们的 8M 是 64KB 的整数倍，所以 size=8M+4KB=8196KB。然后我们可以调用 mmap 了。

```
#if TLS_TCB_AT_TP
      //我们走这个分支，而 pd 将填入 pthread_create 第一个参数指针对应的地址。
      pd = (struct pthread *) ((char *) mem + size - coloring) - 1;
#elif TLS_DTV_AT_TP
      pd = (struct pthread *) ((((uintptr_t) mem + size - coloring
                              - __static_tls_size)
                          & ~__static_tls_align_m1)
                      - TLS_PRE_TCB_SIZE);
#endif


          /* Remember the stack-related values. */
          pd->stackblock = mem;
          pd->stackblock_size = size;

          /* We allocated the first block thread-specific data array.
             This address will not change for the lifetime of this
             descriptor. */
          pd->specific[0] = pd->specific_1stblock;

          /* This is at least the second thread. */
          pd->header.multiple_threads = 1
```

后面有一段 coloring 的代码，完全看不明白，总之了，color 的值决定了 pthread_t 这个返回值的位置。

```
(gdb) p mem
$20 = (void *) 0xb75f9000
(gdb) p pd
$21 = (struct pthread *) 0xb7df9b40
(gdb)
```

接下来的内容就是这几天折磨的哥死去活来的内容了，TLS，传说中的 thread local storage。坦率讲，现在也不懂：

```
        /* Allocate the DTV for this thread. */
          if (_dl_allocate_tls (TLS_TPADJ (pd)) == NULL)
          {
              /* Something went wrong. */
```

```
            assert (errno == ENOMEM);

            /* Free the stack memory we just allocated. */
            (void) munmap (mem, size);

            return errno;
          }
```

thread local storage是个啥意思呢。 请看我们的测试程序：

```
__thread int count = 2222;
__thread unsigned long long count2 ;
static __thread int count3
```

我们child1和child2分别修改了count，count2 count3,但是我们发现线程是并行不悖的，换句话说，每个线程有自己的count/count2/count3，从我的输出截图也可以看出来，child1线程的count地址和child2线程的count地址 不同 。这个效果的原因是我加了__thread关键字。介绍这个TLS之前，我先捏个软柿子。

```
int pthread_key_create(pthread_key_t *key,
void (*destructor) (void *));
int pthread_setspecific(pthread_key_t key, const void *value);
int pthread_getspecific(pthread_key_t key);
int pthread_key_delete(pthread_key_t *key)
```

严格意义上讲，pthread_key_create+pthread_setspecific创建出来的变量也是也是TLS，每个线程也一样具有私有的地址空间，存在各自线程空间里面互不影响，但是这厮的地位明显不如__thread高。原因有二：1 太刻意了，不自然。谁愿意用个变量还得pthread_getspecific,不够cool，我等2B好青年不喜欢这种感觉 2 这种pthread_key_create搞出来的每线程变量个数终究有限。

```
#define PTHREAD_KEY_MAX 1024
```

key这个系列函数是啥意思呢？又是怎么实现的呢？

首先pthread_key_create表示我要占个坑，最多是0～1023。到了真正调用pthread_setspecific的时候，是怎么实现的呢？这时候需要看下struct pthread。我们知道，**pthread_self**返回的就是**struct pthread**的地址。OK我们看下pthread的定义：

```
Struct pthread
{
  union
  {
#if !TLS_DTV_AT_TP
    /* This overlaps the TCB as used for TLS without threads (see tls.h).  */
    tcbhead_t header;   // tcb mean thread control blcok
#else
    struct
    {
      int multiple_threads;
      int gscope_flag;
```

```
# ifndef __ASSUME_PRIVATE_FUTEX
      int private_futex;
# endif
    } header;
#endif
    void *__padding[24];
  };
   ....
  struct pthread_key_data
  {
    ...
    uintptr_t seq;
    void *data;
  } specific_1stblock[PTHREAD_KEY_2NDLEVEL_SIZE]; //PTHREAD_KEY_2NDLEVEL_SIZE=32

  struct pthread_key_data
*specific[PTHREAD_KEY_1STLEVEL_SIZE];//PTHREAD_KEY_1STLEVEL_SIZE=32
  ...
  void *(*start_routine) (void *);
  void *arg;
  ...
  void *stackblock; //mmap 分配的 8192+4=8196KB 的起始地址
  size_t stackblock_size; //8196KB
  size_t guardsize;
  size_t reported_guardsize;
  ...
  struct priority_protection_data *tpp;
}
```

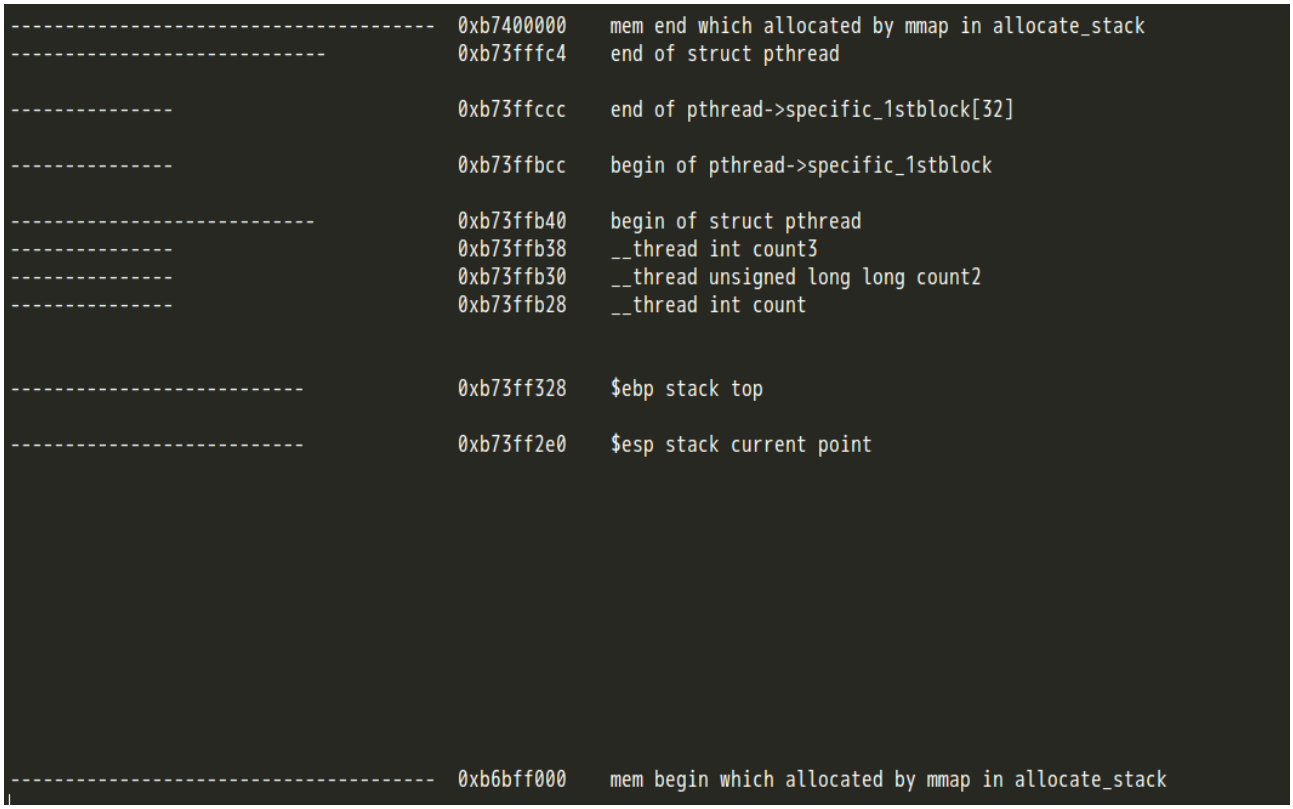
```
allocate_stack 函数：

    /* The first TSD block is included in the TCB.  */
        pd->specific[0] = pd->specific_1stblock;
```

加粗的俩个结构用来实现 pthread_key_XXX 系列函数的。specific[0]这个指针指向 pthread 结构体内部的 specific_1stblock,pthread 结构体里面定义长度为 32 的 pthread_key_data 类型的数组，就像家里有 32 个酒杯。如果 pthread_key_t 类型的变量少于 32 个时候，pthread 结构体里面酒杯就足够。就像家里来的客人少于 32 个，不需要出门买酒杯。很不幸，如果第 33 个客人到来，家里的就就不够了，必须出去买，一次买 32 个回来。注意 32 个酒杯是一组，其中 specific 记录的是每组酒杯的位置。比如我要找第 53 号酒杯，53/32=1 ，第一组（从 0 开始），先从 specific[1]中找到第一组酒杯的位置，然后 53%32=21,从第一组里面找到编号为 21 的酒杯。这种伎俩我们搞 IT 的都比较熟悉。

好，软柿子终于捏完了，该捏核桃了。核桃就是前面提到的 TLS，接口是__thread 关键字。这种方法就自然多了，只要声明是__thread，后面引用变量就像引用普通变量。线程是如何做到的呢？我们

下一篇再讨论。

最后给出一张线程栈的图：



还没讨论的问题有 GS 寄存器是干啥的？ 进程切换（或者 LWP 切换更准确），发生了些什么？ TLS 到底是如何实现的？ 话说 TLS 的确是快硬核桃，我多次试图搞懂多次都失败，今天是不行了，要陪老婆散步去了。

两篇参考文献都非常的好，其中第二篇博客给我的启发最大，正是这篇博文让我鼓起勇气再次探索 TLS，让我这几天痛苦的死去活来。

参考文献

1 Linux 用户空间线程管理介绍之二：创建线程堆栈

2 关于 Linux 线程的线程栈以及 TLS