Linux 线程 之 线程 线程组 进程 轻量级进程（LWP）

Thread Local Storage，线程本地存储，大神 Ulrich Drepper 有篇 PDF 文档是讲 TLS 的，我曾经努力过三次尝试搞清楚 TLS 的原理，均没有彻底搞清楚。这一次是第三次，我沉浸 glibc 的源码和 kernel 的源码中，做了一些实验，也有所得。对 Linux 的线程有了进一步的理解。

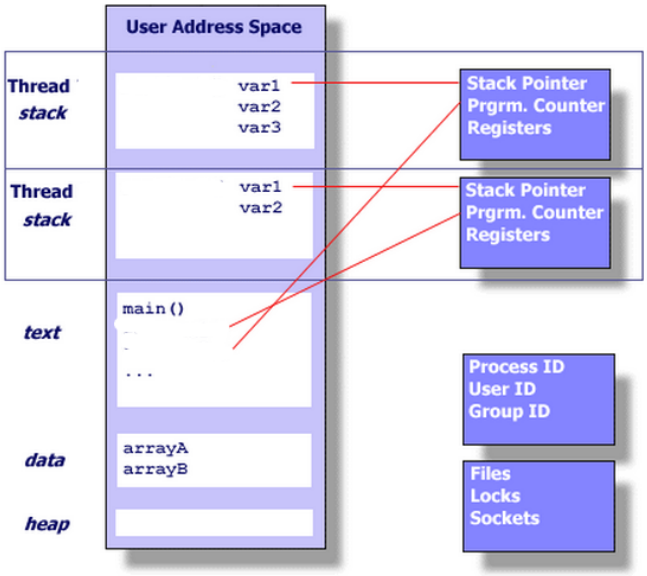
线程是有栈的，我们知道，普通的一个进程，它的栈空间是 8M，我们可以通过 ulmit -a 查看：

```
stack size (kbytes, -s) 8192
```

线程也不例外，线程也是需要栈空间的这句话是废话，呵呵。对于属于同一个进程（或者说是线程组）的多个线程他们是共享一份虚拟内存地址的，如下图所示。这也就决定了，你不能无限制创建线，因为纵然你什么都不做，每个线程默认耗费 8M 的空间（事实上还不止，还有管理结构，后面陈述）。Ulrich Drepper 大神有篇文章《Thread numbers and stacks》，分析了线程栈空间方面的计算。如果我们真的需要很多个线程的话，幸好我们还是可以做一些事情。我们可以通过

pthread_attr_setstacksize,设定好 stack size 属性然后在 pthread_create.

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```



线程栈如上图所示，共享进程（或者称之为线程组）的虚拟地址空间。既然多个线程聚集在一起，我怎么知道我要操作的那个线程栈的地址呢。要解决这个问题，必须要领会线程和进程以及线程组的概念。我不想写一堆片汤话，下面我运行我的测试程序，然后结合现象分析原因：

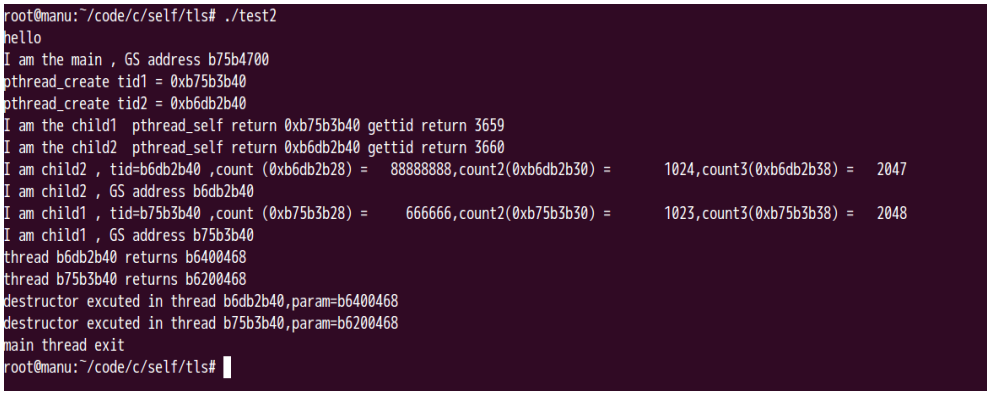
```
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <assert.h>

#define gettid() syscall(__NR_gettid)
```

```
pthread_key_t key;
__thread int count = 2222;
__thread unsigned long long count2 ;
static __thread int count3;
void echomsg(int t)
{
    printf("destructor excuted in thread %x,param=%x\n",pthread_self(),t);
}

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) = %10d,count2(%p) =
%10llu,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 %x\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();
```

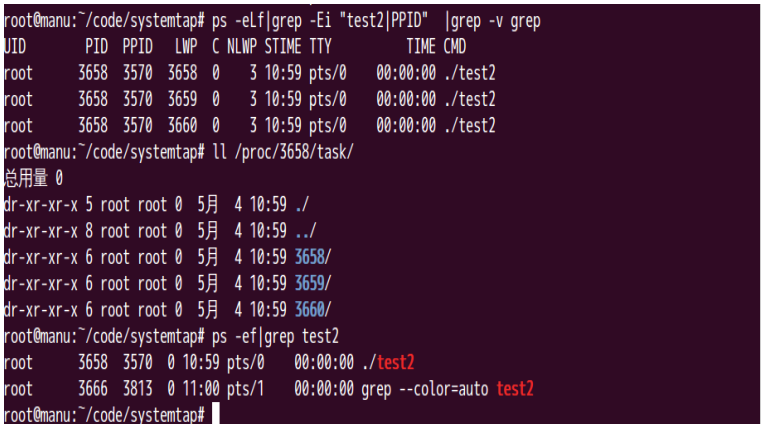
```
    printf("I am the child2 pthread_self return %p gettid return
%d\n",tid,gettid());

    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) = %10d,count2(%p) =
%10llu,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 %x\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;
}
```

这是一个比较综合的程序，因为我下面要多次从不同的侧面分析。对于现在，我们要展示的是进程 线程 线程组的关系。在一个终端运行编译出来的 test2 程序，显示的信息如下



另一个终端看 ps 信息，ps 显示的信息如下：



直接 ps，是看不到我们创建的线程的。只有 3658 一个进程。当我们采用 ps -eLf 的时候，我们看到了三个线程 3658/3659/3660，或者称之为轻量级进程（LWP）。Linux 到底是怎么看待这三者的关系的呢：

Linux 下多线程程序，一般都是有一个主进程通过调用 pthread_create 创建了一个或者多个子线程，如同我们的程序，主进程在 main 中创建了两个子进程。那么 Linux 到底是怎么看待这些事情的呢？

```
pid_t pid;
pid_t tgid;
...
structtask_struct*group_leader;/*threadgroup leader*/
```

上面三个变量是进程描述符的三个成员变量。pid 字面意思是 process id，其实叫 thread id 会更合适。tgid 字面含义是 thread group ID。对于存在多个线程的程序而言，每个线程都有自己的 pid，没错 pid，如同我们例子中的 3658/3659/3660,但是都有个共同的线程组 ID （TGID）：3658 。

好吧，我们再重新说一遍，对于普通进程而言，我们可以称之为只有一个 LWP 的线程组，pid 是它自己的 pid，tgid 还是它自己，线程组里面只有他自己一个光杆司令，自然 group_leader 也是它自己。但是多线程的进程（线程组更恰当）则不然。开天辟地的 main 函数所在的进程会有自己的 PID，也会有也 TGID，group_leader,都是他自己。注意，它自己也是 LWP。后面他使用 ptherad_create 创建了 2 个线程，或者 LWP，这两个新创建的线程会有自己的 PID，但是 TGID 会沿用创建自己的那个进程的 TGID，group_leader 也会尊创建自己的进程的进程描述符（task_struct)为自己的 group_leader。copy_process 函数中有如下代码：

```
p->pid=pid_nr(pid);
p->tgid=p->pid;//普通进程
if(clone_flags &CLONE_THREAD)
p->tgid=current->tgid；//线程选择叫起它的进程的 tgid 作为自己的 tgid
 ....
 p->group_leader = p;//普通进程
 INIT_LIST_HEAD(&p->thread_group);
 ...
 if (clone_flags & CLONE_THREAD) {
  current->signal->nr_threads++;
  atomic_inc(&current->signal->live);
  atomic_inc(&current->signal->sigcnt);
  p->group_leader = current->group_leader;//线程选择叫起它的进程作为它的 group_leader
  list_add_tail_rcu(&p->thread_group, &p->group_leader->thread_group);
}
```

OK,ps -eLf 中有个字段叫 NLWP，就是线程组中 LWP 的个数，对于我们的例子，main 函数所在 LWP+两个线程 = 3.

我们传说的 getpid 函数，本质取得是进程描述符的 TGID，而 gettid 系统调用，取得才是每个 LWP 各自的 PID。请看上面的图片输出，上面连个线程 gettid 返回的是 3873 和 3874,是自己的 PID。稍微有点毁三观。

除此外，需要指出的是用户态 pthread_create 出来的线程，在内核态，也拥有自己的进程描述符 task_struct（copy_process 里面调用 dup_task_struct 创建）。这是什么意思呢。意思是我们用户态所说的线程，一样是内核进程调度的实体。进程调度，严格意义上说应该叫 LWP 调度，进程调度，不是以前面提到的线程组为单位调度的，本质是以 LWP 为单位调度的。这个结论乍一看惊世骇俗，细细一想，其是很合理。我们为什么多线程？因为多 CPU，多核，我们要充分利用多核,同一个线程组的不同 LWP 是可以同时跑在不同的 CPU 之上的，因为这个并发，所以我们有线程锁的设计，这从侧面证明了，

LWP 是调度的实体。

我们用 systemtap 去观察下 test2 程序相关的调度：systemtap 脚本如下：

```
#!/usr/bin/env stap
#
#
global time_offset

probe begin
{
    time_offset=gettimeofday_us()
    printf("monitor begin==========\n")
}
probe scheduler.cpu_off
{
    if(task_execname(task_next)=="test2")
    {
        t=gettimeofday_us();
        printf("%9d : %20s(%6d)->%10s(%6d:%6d)\n",
            t-time_offset,
            task_execname(task_prev),
            task_pid(task_prev),
            task_execname(task_next),
            task_pid(task_next),  #返回的是内核中的 TGID
            task_tid(task_next))  #返回的内核中的 PID
    }
}
```

我们的二进制可执行程序叫做 test2，一个终端叫起 systemtap，另一个终端叫起 test2,查看下输出：

```
root@manu:~/code/systemtap# stap switch.stp
monitor begin==========
  4626628 :              swapper/3(     0)->     test2(  3790:  3791)
  4626628 :              swapper/1(     0)->     test2(  3790:  3792)
  4626714 :              swapper/1(     0)->     test2(  3790:  3792)
  4626823 :              swapper/0(     0)->     test2(  3790:  3791)
  4626855 :              swapper/1(     0)->     test2(  3790:  3792)
  4626887 :              swapper/0(     0)->     test2(  3790:  3791)
  4626981 :              swapper/1(     0)->     test2(  3790:  3792)
  4627018 :              swapper/0(     0)->     test2(  3790:  3791)
  4627108 :              swapper/1(     0)->     test2(  3790:  3792)
  4627152 :              swapper/0(     0)->     test2(  3790:  3791)
  4627191 :              swapper/1(     0)->     test2(  3790:  3792)
  4627244 :              swapper/1(     0)->     test2(  3790:  3792)
  5627373 :              swapper/1(     0)->     test2(  3790:  3792)
  6627328 :              swapper/0(     0)->     test2(  3790:  3791)
 55627550 :              swapper/1(     0)->     test2(  3790:  3792)
 56627510 :              swapper/0(     0)->     test2(  3790:  3791)
 64626689 :              swapper/2(     0)->     test2(  3790:  3790)
```

```
root@manu:~/code/c/self/tls# ./test2
hello
I am the main , GS address b7587700
pthread_create tid1 = 0xb7586b40
pthread_create tid2 = 0xb6d85b40
I am the child1  pthread_self return 0xb7586b40 gettid return 3791
I am the child2  pthread_self return 0xb6d85b40 gettid return 3792
I am child1 , tid=b7586b40 ,count (0xb7586b28) =     666666,count2(0xb7586b30) =       1023,count3(0xb7586b38) =   2048
I am child1 , GS address b7586b40
I am child2 , tid=b6d85b40 ,count (0xb6d85b28) =   88888888,count2(0xb6d85b30) =       1024,count3(0xb6d85b38) =   2047
I am child2 , GS address b6d85b40
thread b6d85b40 returns b6400468
thread b7586b40 returns b6200468
destructor excuted in thread b6d85b40,param=b6400468
destructor excuted in thread b7586b40,param=b6200468
main thread exit
root@manu:~/code/c/self/tls#
```

上面三个 LWP 都是 CPU 友好型的，如果同属一个线程组的多个线程（或者称之为 LWP）都是 CPU 消耗型，你可以看到激烈的争夺 CPU 资源。

本想继续写下去，无奈太长了，不想变成滚轮杀手，在下一篇写其他内容吧。参考文献提到的文章，非常的好，甚至提到了线程组里面信号的处理，信号不是我这篇博文的重点，所以我略过不提了。

参考文献

1　Linux 2.6 内核中的线程组初探（好文章，强烈推荐）