

# 10. 线程概念与控制

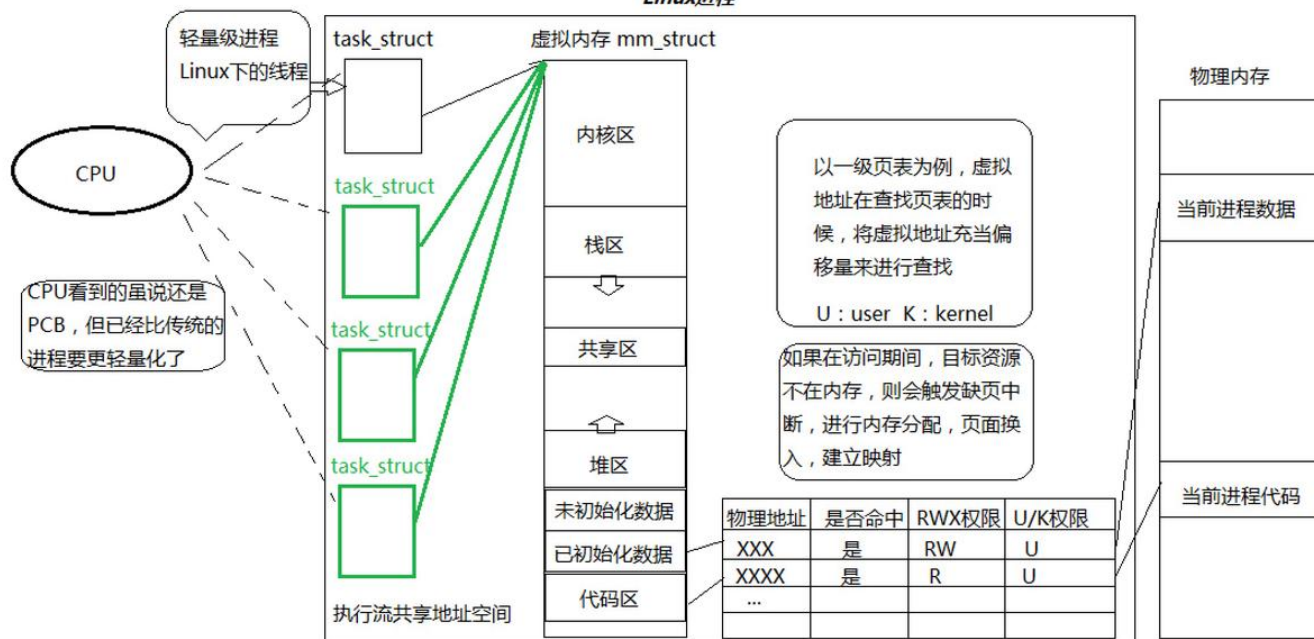
## 本节重点：

1. 深刻理解线程
2. 深刻理解虚拟地址空间
3. 了解线程概念，理解线程与进程区别与联系。
4. 学会线程控制，线程创建，线程终止，线程等待。
5. 了解线程分离与线程安全概念。
6. 掌握线程与进程地址空间布局
7. 理解LWP和原生线程库封装关系

## 1. Linux线程概念

### 1-1 什么是线程

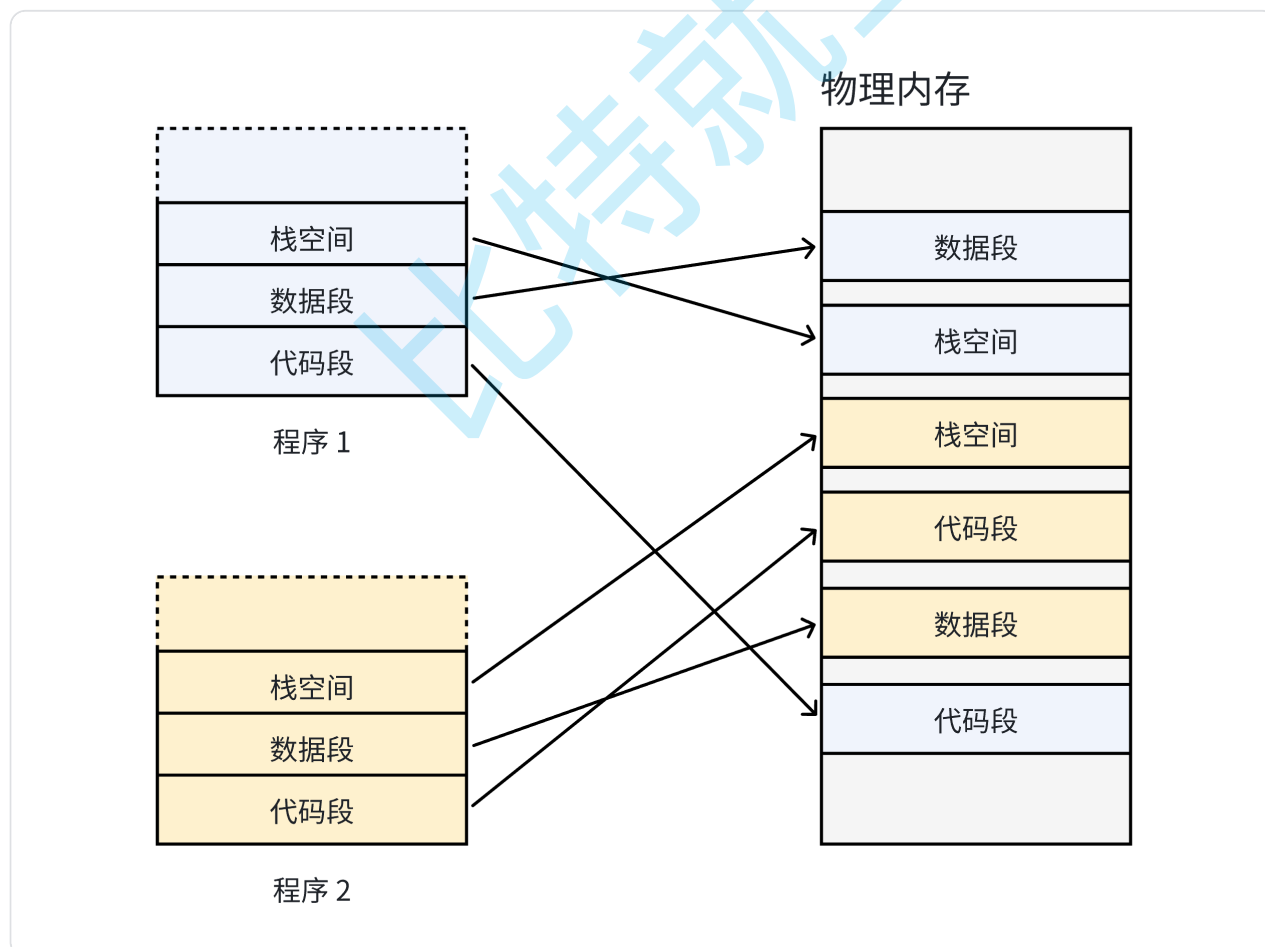
- 在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列”
- 一切进程至少都有一个执行线程
- 线程在进程内部运行，本质是在进程地址空间内运行
- 在Linux系统中，在CPU眼中，看到的PCB都要比传统的进程更加轻量化
- 透过进程虚拟地址空间，可以看到进程的大部分资源，将进程资源合理分配给每个执行流，就形成了线程执行流



## 1-2 分页式存储管理

### 1-2-1 虚拟地址和页表的由来

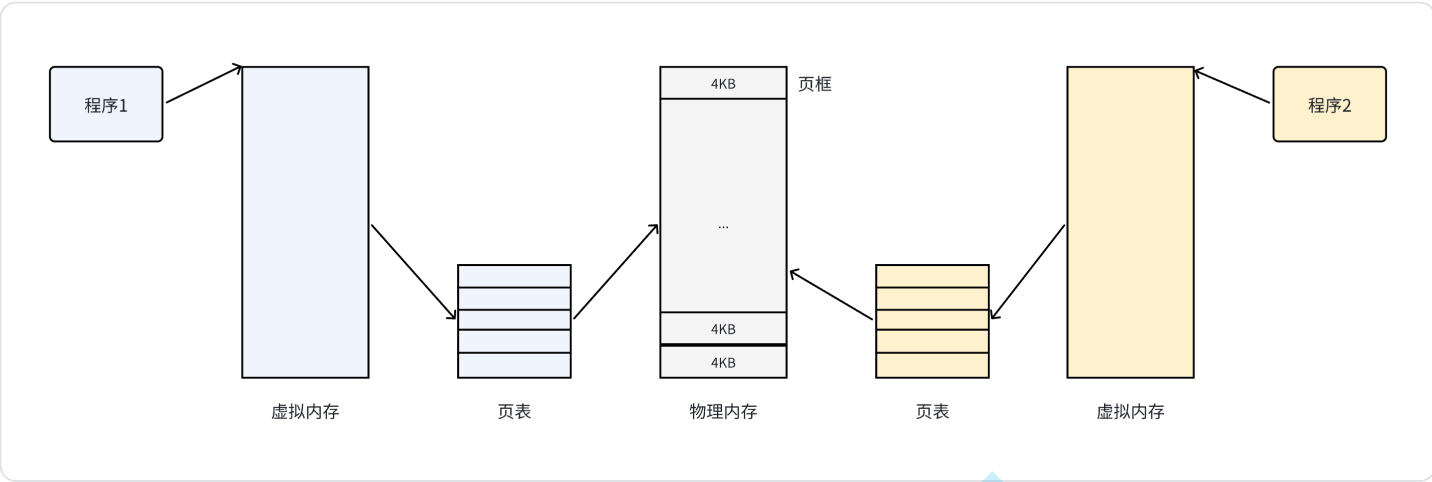
思考一下，如果在没有虚拟内存和分页机制的情况下，每一个用户程序在物理内存上所对应的空间必须是连续的，如下图：



因为每一个程序的代码、数据长度都是不一样的，按照这样的映射方式，物理内存将会被分割成各种离散的、大小不同的块。经过一段运行时间之后，有些程序会退出，那么它们占据的物理内存空间可

以被回收，导致这些物理内存都是以很多碎片的形式存在。

怎么办呢？我们希望操作系统提供给用户的空间必须是连续的，但是物理内存最好不要连续。此时虚拟内存和分页便出现了，如下图所示：



把物理内存按照一个固定的长度的页框进行分割，有时叫做物理页。每个页框包含一个物理页（page）。一个页的大小等于页框的大小。大多数 32 位 体系结构支持 4KB 的页，而 64 位 体系结构一般会支持 8KB 的页。区分一页和一个页框是很重要的：

- 页框是一个存储区域；
- 而页是一个数据块，可以存放在任何页框或磁盘中。

有了这种机制，CPU 便并非直接访问物理内存地址，而是通过虚拟地址空间来间接的访问物理内存地址。所谓的虚拟地址空间，是操作系统为每一个正在执行的进程分配的一个逻辑地址，在32位机上，其范围从0 ~ 4G-1。

操作系统通过将虚拟地址空间和物理内存地址之间建立映射关系，也就是页表，这张表上记录了每一对页和页框的映射关系，能让CPU间接的访问物理内存地址。

总结一下，其思想是将虚拟内存下的逻辑地址空间分为若干页，将物理内存空间分为若干页框，通过页表便能把连续的虚拟内存，映射到若干个不连续的物理内存页。这样就解决了使用连续的物理内存造成的碎片问题。

### 1-2-2 物理内存管理

假设一个可用的物理内存有 4GB 的空间。按照一个页框的大小 4KB 进行划分，4GB 的空间就是  $4GB/4KB = 1048576$  个页框。有这么多的物理页，操作系统肯定是要将其管理起来的，操作系统需要知道哪些页正在被使用，哪些页空闲等等。

内核用 struct page 结构表示系统中的每个物理页，出于节省内存的考虑，struct page 中使用了大量的联合体union。

```
1 /* include/linux/mm_types.h */
2 struct page {
3     /* 原子标志，有些情况下会异步更新 */
```

```

4     unsigned long flags;
5     union {
6         struct {
7             /* 换出页列表, 例如由zone->lru_lock保护的active_list */
8             struct list_head lru;
9             /* 如果最低为0, 则指向inode
10              * address_space, 或为NULL
11              * 如果页映射为匿名内存, 最低为置位
12              * 而且该指针指向anon_vma对象
13              */
14             struct address_space* mapping;
15             /* 在映射内的偏移量 */
16             pgoff_t index;
17             /*
18              * 由映射私有, 不透明数据
19              * 如果设置了PagePrivate, 通常用于buffer_heads
20              * 如果设置了PageSwapCache, 则用于swp_entry_t
21              * 如果设置了PG_buddy, 则用于表示伙伴系统中的阶
22              */
23             unsigned long private;
24         };
25         struct { /* slab, slob and slub */
26             union {
27                 struct list_head slab_list; /* uses lru */
28                 struct { /* Partial pages */
29                     struct page* next;
30 #ifdef CONFIG_64BIT
31                     int pages; /* Nr of pages left */
32                     int pobjects; /* Approximate count */
33 #else
34                     short int pages;
35                     short int pobjects;
36 #endif
37                 };
38             };
39             struct kmem_cache* slab_cache; /* not slob */
40             /* Double-word boundary */
41             void* freelist; /* first free object */
42             union {
43                 void* s_mem; /* slab: first object */
44                 unsigned long counters; /* SLUB */
45                 struct { /* SLUB */
46                     unsigned inuse : 16; /* 用于SLUB分配器: 对象的数目 */
47                     unsigned objects : 15;
48                     unsigned frozen : 1;
49                 };
50             };

```

```

51     };
52     ...
53 };
54
55 union {
56     /* 内存管理子系统中映射的页表项计数，用于表示页是否已经映射，还用于限制逆向映射搜索*/
57     atomic_t _mapcount;
58     unsigned int page_type;
59     unsigned int active;          /* SLAB */
60     int units;                   /* SLOB */
61 };
62 ...
63 #if defined(WANT_PAGE_VIRTUAL)
64     /* 内核虚拟地址（如果没有映射则为NULL，即高端内存） */
65     void* virtual;
66 #endif /* WANT_PAGE_VIRTUAL */
67 ...
68 }

```

其中比较重要的几个参数：

1. `flags`：用来存放页的状态。这些状态包括页是不是脏的，是不是被锁定在内存中等。flag的每一位单独表示一种状态，所以它至少可以同时表示出32种不同的状态。这些标志定义在<linux/page-flags.h>中。其中一些比特位非常重要，如PG\_locked用于指定页是否锁定，PG\_uptodate用于表示页的数据已经从块设备读取并且没有出现错误。
2. `_mapcount`：表示在页表中有多少项指向该页，也就是这一页被引用了多少次。当计数值变为-1时，就说明当前内核并没有引用这一页，于是在新的分配中就可以使用它。
3. `virtual`：是页的虚拟地址。通常情况下，它就是页在虚拟内存中的地址。有些内存（即所谓的高端内存）并不永久地映射到内核地址空间上。在这种情况下，这个域的值NULL，需要的时候，必须动态地映射这些页。

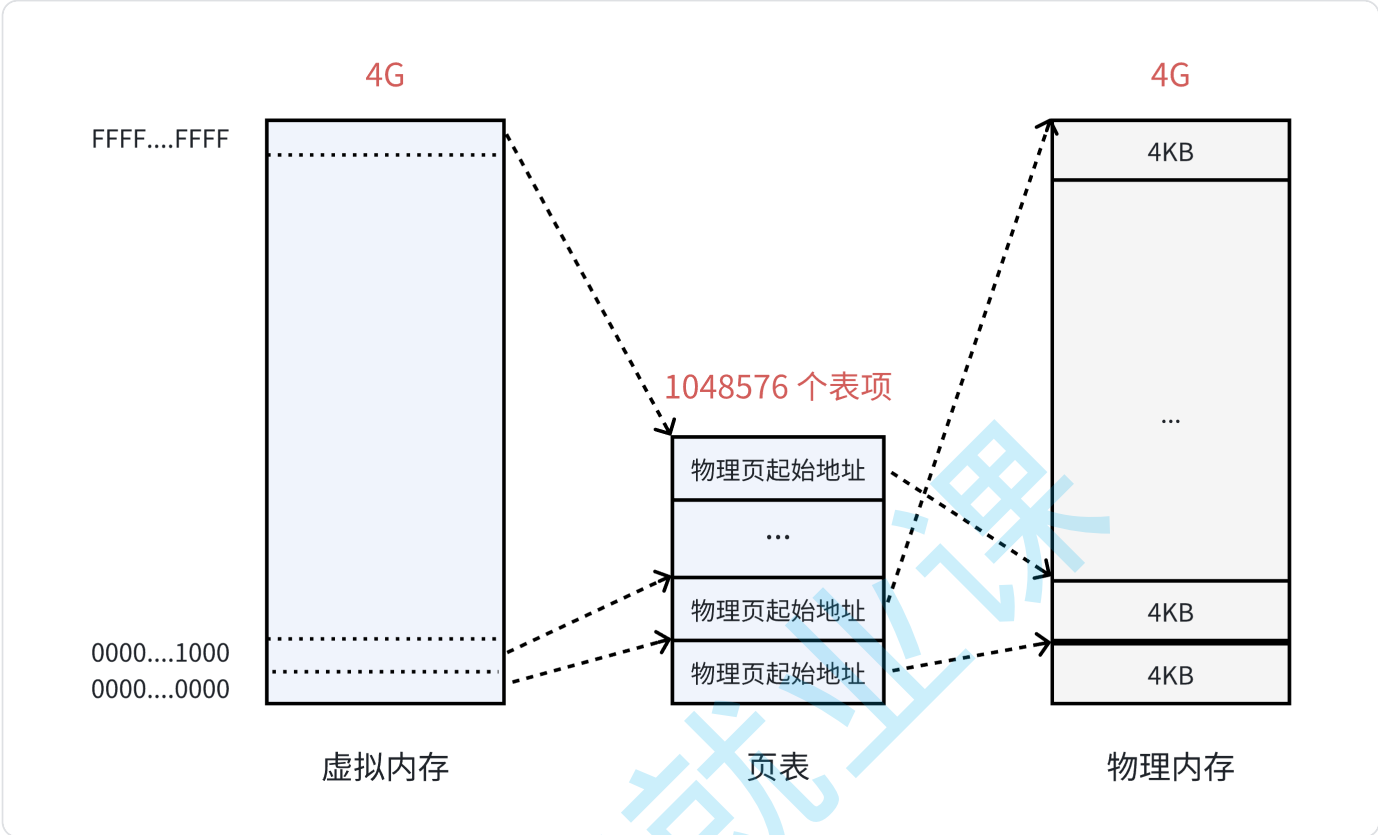
要注意的是 `struct page` 与物理页相关，而并非与虚拟页相关。而系统中的每个物理页都要分配一个这样的结构体，让我们来算算对所有这些页都这么做，到底要消耗掉多少内存。

算 `struct page` 占40个字节的内存吧，假定系统的物理页为 4KB 大小，系统有 4GB 物理内存。那么系统中共有页面 1048576 个（1兆个），所以描述这么多页面的page结构体消耗的内存只不过 40MB，相对系统 4GB 内存而言，仅是很小的一部分罢了。因此，要管理系统中这么多物理页面，这个代价并不算太大。

要知道的是，页的大小对于内存利用和系统开销来说非常重要，页太大，页页必然会剩余较大不能利用的空间（页内碎片）。页太小，虽然可以减小页内碎片的大小，但是页太多，会使得页表太长而占用内存，同时系统频繁地进行页转化，加重系统开销。因此，页的大小应该适中，通常为 512B - 8KB，windows系统的页框大小为4KB。

1-2-3 页表

页表中的每一个表项，指向一个物理页的开始地址。在 32 位系统中，虚拟内存的最大空间是 4GB，这是每一个用户程序都拥有的虚拟内存空间。既然需要让 4GB 的虚拟内存全部可用，那么页表中就需要能够表示这所有的 4GB 空间，那么就一共需要  $4GB/4KB = 1048576$  个表项。如下图所示：



虚拟内存看上去被虚线“分割”成一个个单元，其实并不是真的分割，虚拟内存仍然是连续的。这个虚线的单元仅仅表示它与页表中每一个表项的映射关系，并最终映射到相同大小的一个物理内存页上。

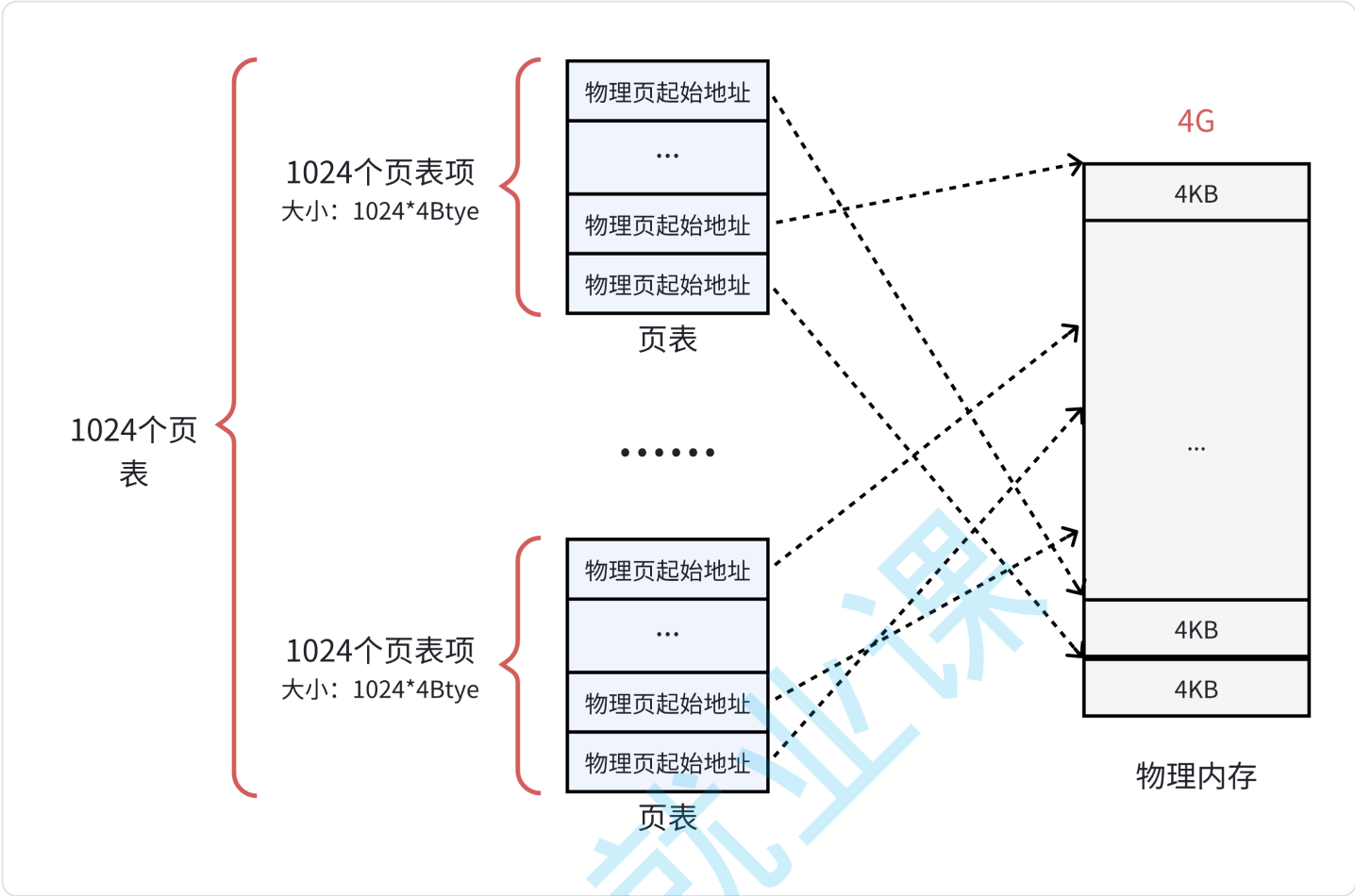
页表中的物理地址，与物理内存之间，是随机的映射关系，哪里可用就指向哪里(物理页)。虽然最终使用的物理内存是离散的，但是与虚拟内存对应的线性地址是连续的。处理器在访问数据、获取指令时，使用的都是线性地址，只要它是连续的就可以了，最终都能够通过页表找到实际的物理地址。

在 32 位系统中，地址的长度是 4 个字节，那么页表中的每一个表项就是占用 4 个字节。所以页表占据的总空间大小就是： $1048576 \times 4 = 4MB$  的大小。也就是说映射表自己本身，就要占用  $4MB / 4KB = 1024$  个物理页。这会存在哪些问题呢？

- 回想一下，当初为什么使用页表，就是要将进程划分为一个个页可以不用连续的存放在物理内存中，但是此时页表就需要1024个连续的页框，似乎和当时的目标有点背道而驰了.....
- 此外，根据局部性原理可知，很多时候进程在一段时间内只需要访问某几个页就可以正常运行了。因此也没有必要一次让所有的物理页都常驻内存。

解决需要大容量页表的最好方法是：把页表看成普通的文件，对它进行离散分配，即对页表再分页，由此形成多级页表的思想。

为了解决这个问题，可以把这个单一页表拆分成 1024 个体积更小的映射表。如下图所示。这样一来， $1024(\text{每个表中的表项个数}) * 1024(\text{表的个数})$ ，仍然可以覆盖 4GB 的物理内存空间。



这里的每一个表，就是真正的页表，所以一共有 1024 个页表。一个页表自身占用 4KB ，那么 1024 个页表一共就占用了 4MB 的物理内存空间，和之前没差别啊？

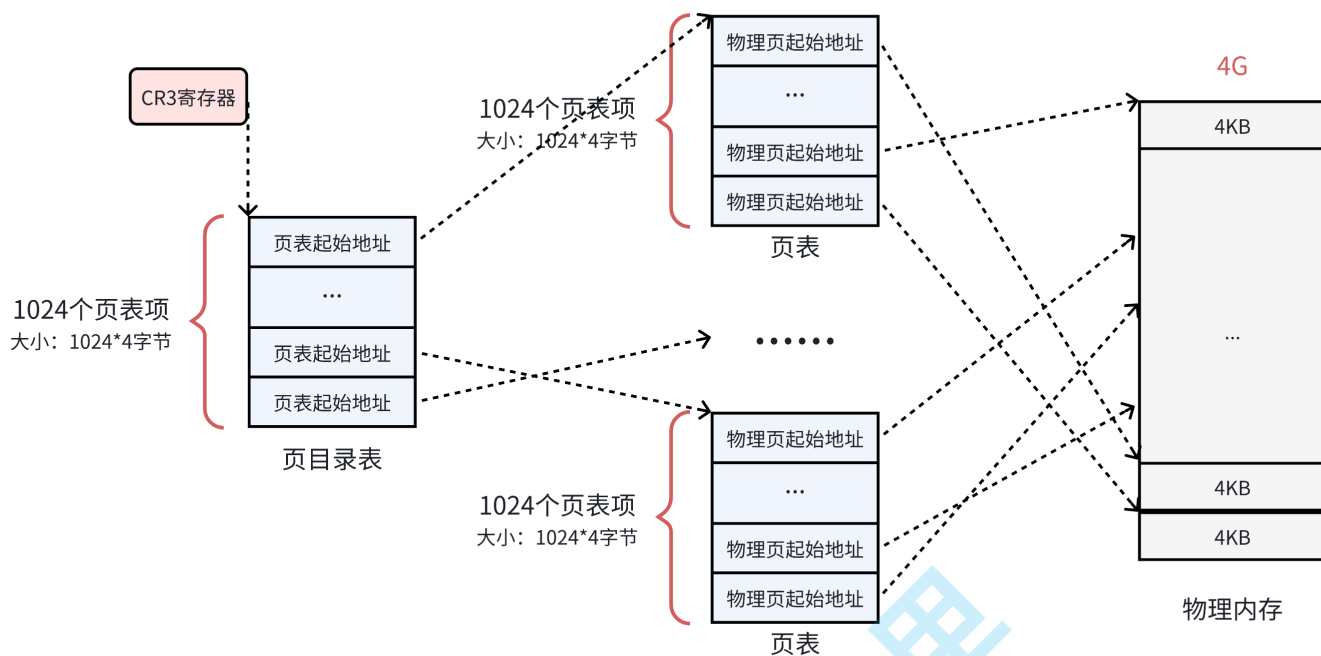
从总数上看是这样，但是一个应用程序是不可能完全使用全部的 4GB 空间的，也许只要几十个页表就可以了。例如：一个用户程序的代码段、数据段、栈段，一共就需要 10 MB 的空间，那么使用 3 个页表就足够了。

计算过程：  
每一个页表项指向一个 4KB 的物理页，那么一个页表中 1024 个页表项，一共能覆盖 4MB 的物理内存；  
那么 10MB 的程序，向上对齐取整之后(4MB 的倍数，就是 12 MB)，就需要 3 个页表就可以了。

### 1-2-4 页目录结构

到目前为止，每一个页框都被一个页表中的一个表项来指向了，那么这 1024 个页表也需要被管理起来。管理页表的表称之为**页目录表**，形成二级页表。如下图所示：





- 所有页表的物理地址被页目录表项指向
- 页目录的物理地址被 **CR3 寄存器** 指向，这个寄存器中，保存了当前正在执行任务的页目录地址。

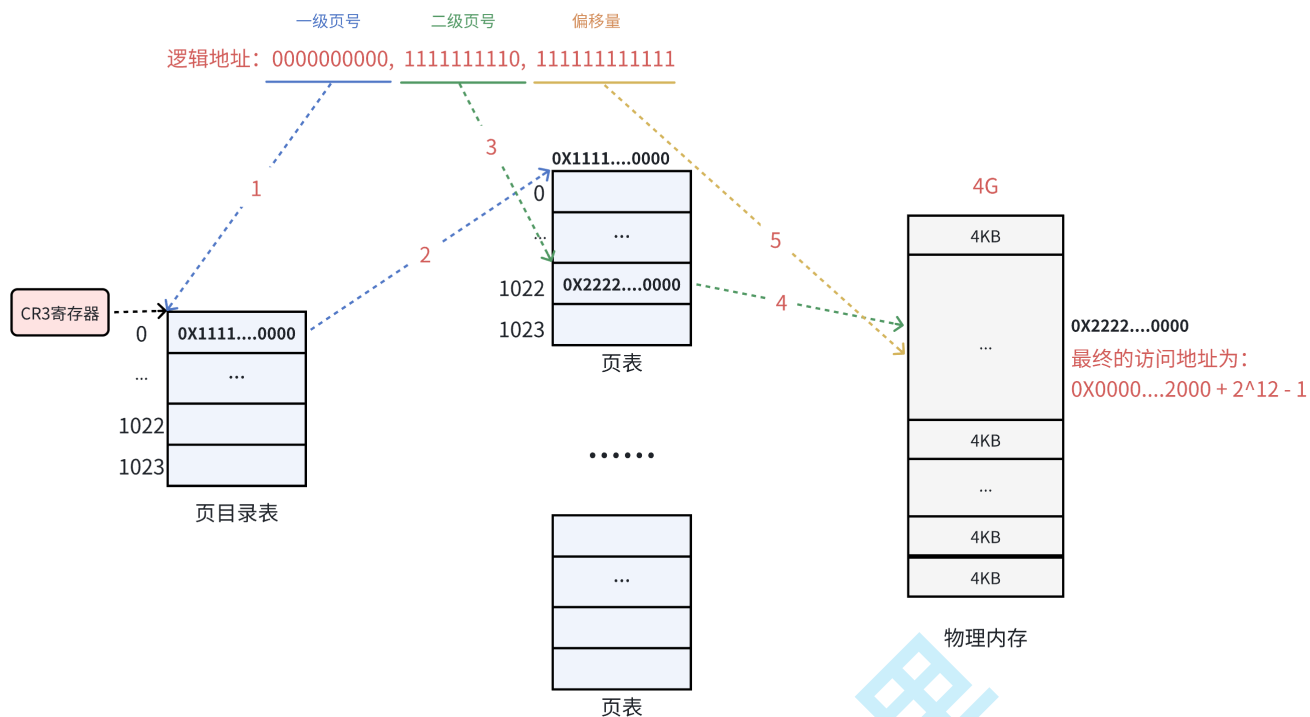
所以操作系统在加载用户程序的时候，不仅仅需要为程序内容来分配物理内存，还需要为用来保存程序的页目录和页表分配物理内存。

### 1-2-5 两级页表的地址转换

下面以一个逻辑地址为例。将逻辑地址（`00000000000,0000000001,1111111111`）转换为物理地址的过程：

1. 在32位处理器中，采用4KB的页大小，则虚拟地址中低12位为页偏移，剩下高20位给页表，分成两级，每个级别占10个bit（10+10）。
2. **CR3 寄存器** 读取页目录起始地址，再根据一级页号查页目录表，找到下一级页表在物理内存中存放位置。
3. 根据二级页号查表，找到最终想要访问的内存块号。
4. 结合页内偏移量得到物理地址。





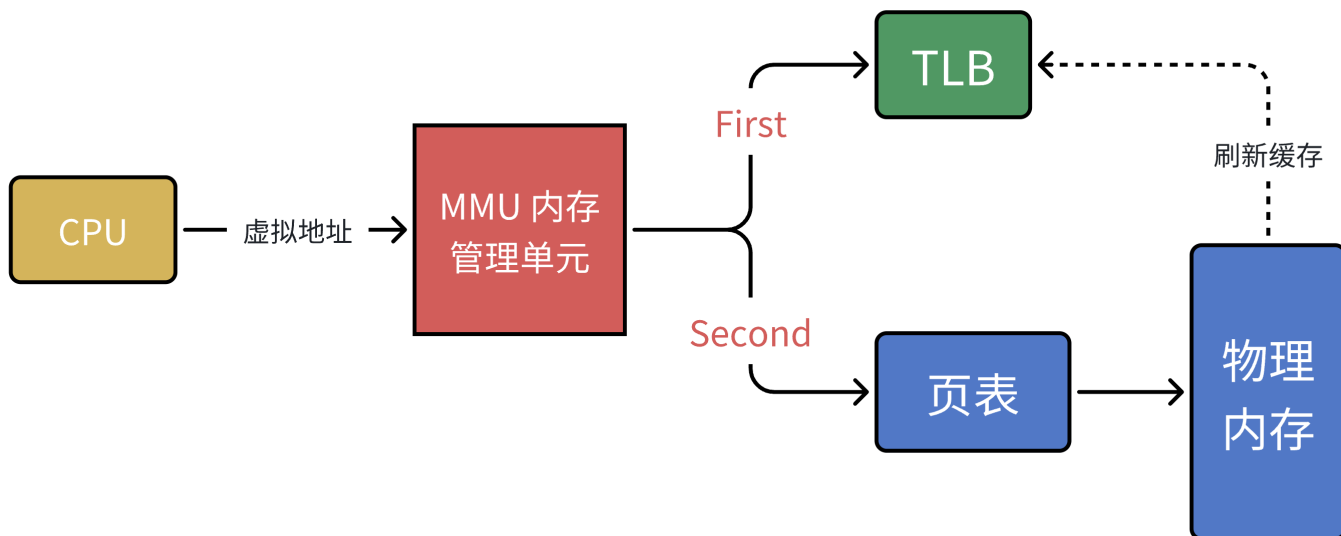
5. 注：一个物理页的地址一定是 4KB 对齐的(最后的 12 位全部为 0)，所以其实只需要记录物理页地址的高 20 位即可。
6. 以上其实就是 MMU 的工作流程。MMU(Memory Manage Unit)是一种硬件电路，其速度很快，主要工作是进行内存管理，地址转换只是它承接的业务之一。

到这里其实还有个问题，MMU要先进行两次页表查询确定物理地址，在确认了权限等问题后，MMU再将这个物理地址发送到总线，内存收到之后开始读取对应地址的数据并返回。那么当页表变为N级时，就变成了N次检索+1次读写。可见，页表级数越多查询的步骤越多，对于CPU来说等待时间越长，效率越低。

让我们现在总结一下：**单级页表对连续内存要求高，于是引入了多级页表，但是多级页表也是一把双刃剑，在减少连续存储要求且减少存储空间的同时降低了查询效率。**

有没有提升效率的办法呢？计算机科学中的所有问题，都可以通过添加一个中间层来解决。MMU 引入了新武器，江湖人称快表的 **TLB**（其实，就是缓存）

当 CPU 给 MMU 传新虚拟地址之后，MMU 先去问 TLB 那边有没有，如果有就直接拿到物理地址发到总线给内存，齐活。但 TLB 容量比较小，难免发生 **Cache Miss**，这时候 MMU 还有保底的老武器页表，在页表中找到之后 MMU 除了把地址发到总线传给内存，还把这条映射关系给到 TLB，让它记录一下刷新缓存。

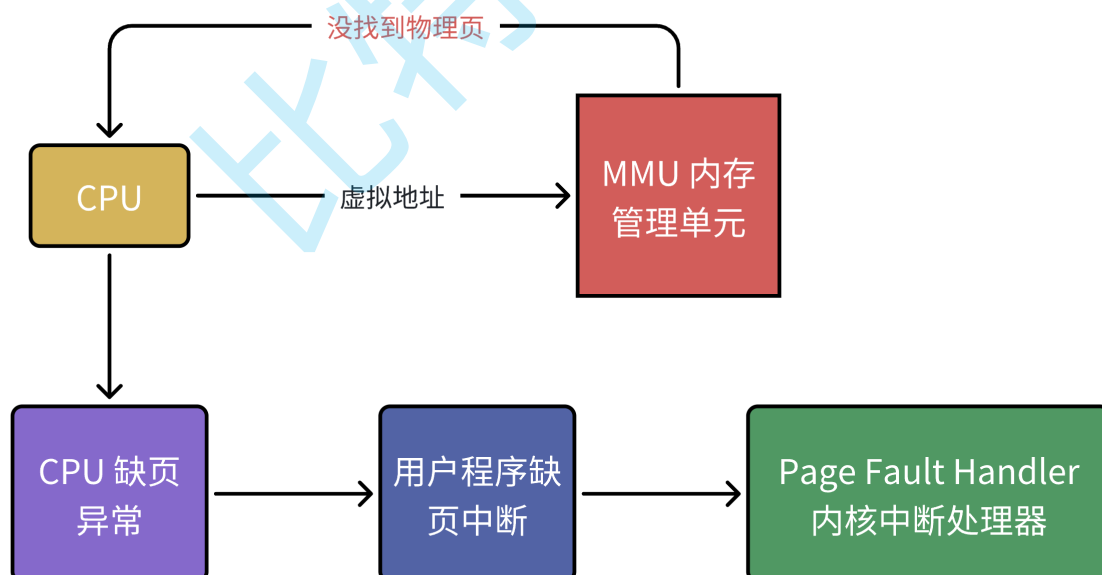


### 1-2-6 缺页异常

设想，CPU 给 MMU 的虚拟地址，在 TLB 和页表都没有找到对应的物理页，该怎么办呢？其实这就是缺页异常 `Page Fault`，它是一个由硬件中断触发的可以由软件逻辑纠正的错误。

假如目标内存页在物理内存中没有对应的物理页或者存在但无对应权限，CPU 就无法获取数据，这种情况下CPU就会报告一个缺页错误。

由于 CPU 没有数据就无法进行计算，CPU罢工了用户进程也就出现了缺页中断，进程会从用户态切换到内核态，并将缺页中断交给内核的 `Page Fault Handler` 处理。



缺页中断会交给 `PageFaultHandler` 处理，其根据缺页中断的不同类型会进行不同的处理：

- `Hard Page Fault` 也被称为 `Major Page Fault`，翻译为硬缺页错误/主要缺页错误，这时物理内存中没有对应的物理页，需要CPU打开磁盘设备读取到物理内存中，再让MMU建立虚拟地址和物理地址的映射。

- `Soft Page Fault` 也被称为 `Minor Page Fault`，翻译为软缺页错误/次要缺页错误，这时物理内存中是存在对应物理页的，只不过可能是其他进程调入的，发出缺页异常的进程不知道而已，此时MMU只需要建立映射即可，无需从磁盘读取写入内存，一般出现在多进程共享内存区域。
- `Invalid Page Fault` 翻译为无效缺页错误，比如进程访问的内存地址越界访问，又比如对空指针解引用内核就会报 `segment fault` 错误中断进程直接挂掉。

### 1-3 线程的优点

- 创建一个新线程的代价要比创建一个新进程小得多
- 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多
  - 最主要的区别是**线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的**。这两种上下文切换的处理都是通过操作系统内核来完成的。内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出。
  - 另外一个隐藏的损耗是**上下文的切换会扰乱处理器的缓存机制**。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲 `TLB`（快表）会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题，当然还有硬件cache。
- 线程占用的资源要比进程少很多
- 能充分利用多处理器的可并行数量
- 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务
- 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现
- I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

### 1-4 线程的缺点

- 性能损失
  - 一个很少被外部事件阻塞的计算密集型线程往往无法与其它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。
- 健壮性降低
  - 编写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。
- 缺乏访问控制
  - 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。
- 编程难度提高

- 编写与调试一个多线程程序比单线程程序困难得多

## 1-5 线程异常

- 单个线程如果出现除零，野指针问题导致线程崩溃，进程也会随着崩溃
- 线程是进程的执行分支，线程出异常，就类似进程出异常，进而触发信号机制，终止进程，进程终止，该进程内的所有线程也就随即退出

## 1-6 线程用途

- 合理的使用多线程，能提高CPU密集型程序的执行效率
- 合理的使用多线程，能提高IO密集型程序的用户体验（如生活中我们一边写代码一边下载开发工具，就是多线程运行的一种表现）

# 2. Linux进程VS线程

## 2-1 进程和线程

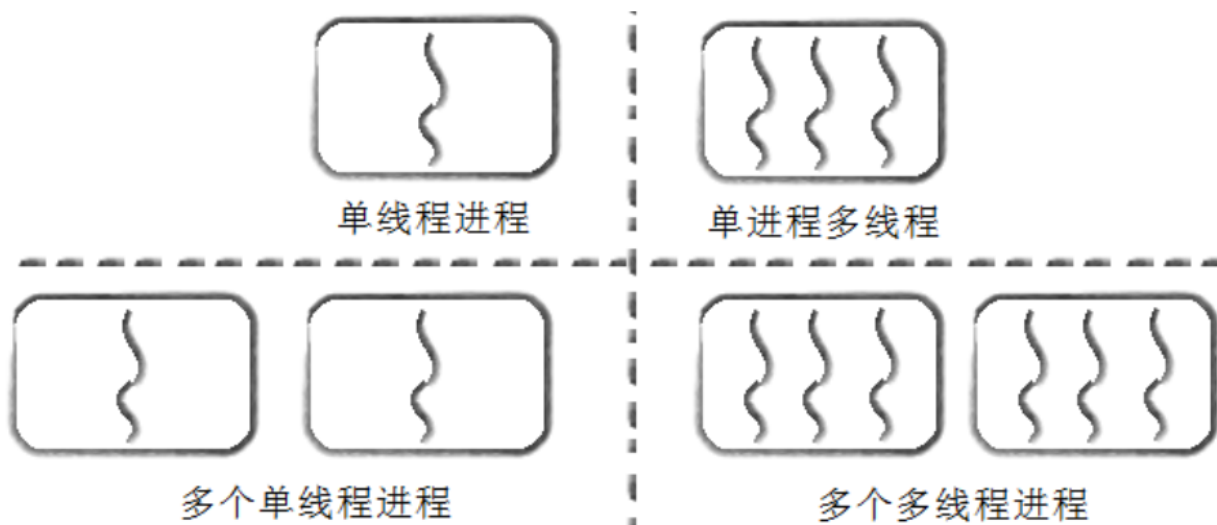
- 进程是资源分配的基本单位
- 线程是调度的基本单位
- 线程共享进程数据，但也拥有自己的一部分数据：
  - 线程ID
  - 一组寄存器
  - 栈
  - errno
  - 信号屏蔽字
  - 调度优先级

## 2-2 进程的多个线程共享

同一地址空间,因此Text Segment、Data Segment都是共享的,如果定义一个函数,在各线程中都可以调用,如果定义一个全局变量,在各线程中都可以访问到,除此之外,各线程还共享以下进程资源和环境:

- 文件描述符表
- 每种信号的处理方式(SIG\_IGN、SIG\_DFL或者自定义的信号处理函数)
- 当前工作目录
- 用户id和组id

进程和线程的关系如下图:



## 2-3 关于进程线程的问题

- 如何看待之前学习的单进程？具有一个线程执行流的进程

## 3. Linux线程控制

### 3-1 POSIX线程库

- 与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread\_”打头的
- 要使用这些函数库，要通过引入头文 `<pthread.h>`
- 链接这些线程函数库时要使用编译器命令的“-lpthread”选项

### 3-2 创建线程

```
1 功能：创建一个新的线程
2 原型：
3     int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
      (*start_routine)(void*), void *arg);
4
5 参数：
6     thread:返回线程ID
7     attr:设置线程的属性，attr为NULL表示使用默认属性
8     start_routine:是个函数地址，线程启动后要执行的函数
9     arg:传给线程启动函数的参数
10
11 返回值：成功返回0；失败返回错误码
```

#### 错误检查:

- 传统的一些函数是，成功返回0，失败返回-1，并且对全局变量errno赋值以指示错误。

- pthreads函数出错时不会设置全局变量errno（而大部分其他POSIX函数会这样做）。而是将错误代码通过返回值返回
- pthreads同样也提供了线程内的errno变量，以支持其它使用errno的代码。对于pthreads函数的错误，建议通过返回值判定，因为读取返回值要比读取线程内的errno变量的开销更小

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <pthread.h>
6
7 void *rout(void *arg) {
8     int i;
9     for(;;) {
10         printf("I'am thread 1\n");
11         sleep(1);
12     }
13 }
14
15 int main( void )
16 {
17     pthread_t tid;
18     int ret;
19     if ( (ret=pthread_create(&tid, NULL, rout, NULL)) != 0 ) {
20         fprintf(stderr, "pthread_create : %s\n", strerror(ret));
21         exit(EXIT_FAILURE);
22     }
23
24     int i;
25     for(;;) {
26         printf("I'am main thread\n");
27         sleep(1);
28     }
29 }
```

```
1 #include <pthread.h>
2
3 // 获取线程ID
4 pthread_t pthread_self(void);
```

打印出来的 tid 是通过 pthread 库中有函数 `pthread_self` 得到的，它返回一个 pthread\_t 类型的变量，指代的是调用 pthread\_self 函数的线程的 “ID”。

怎么理解这个“ID”呢？这个“ID”是 pthread 库给每个线程定义的进程内唯一标识，是 pthread 库维持的。

由于每个进程有自己独立的内存空间，故此“ID”的作用域是进程级而非系统级（内核不认识）。

其实 pthread 库也是通过内核提供的系统调用（例如clone）来创建线程的，而内核会为每个线程创建系统全局唯一的“ID”来唯一标识这个线程。

### 使用PS命令查看线程信息

运行代码后执行：

```
1 $ ps -aL | head -1 && ps -aL | grep mythread
2      PID      LWP TTY          TIME CMD
3 2711838 2711838 pts/235    00:00:00 mythread
4 2711838 2711839 pts/235    00:00:00 mythread
5
6 -L 选项：打印线程信息
```

LWP 是什么呢？LWP 得到的是真正的线程ID。之前使用 `pthread_self` 得到的这个数实际上是一个地址，在虚拟地址空间上的一个地址，通过这个地址，可以找到关于这个线程的基本信息，包括线程ID，线程栈，寄存器等属性。

在 `ps -aL` 得到的线程ID，有一个线程ID和进程ID相同，这个线程就是主线程，主线程的栈在虚拟地址空间的栈上，而其他线程的栈是在共享区（堆栈之间），因为pthread系列函数都是pthread库提供给我们的。而pthread库是在共享区的。所以除了主线程之外的其他线程的栈都在共享区。

## 3-3 线程终止

如果需要只终止某个线程而不终止整个进程,可以有三种方法:

1. 从线程函数return。这种方法对主线程不适用,从main函数return相当于调用exit。
2. 线程可以调用pthread\_exit终止自己。
3. 一个线程可以调用pthread\_cancel终止同一进程中的另一个线程。

### pthread\_exit函数

```
1 功能：线程终止
2
3 原型：
4     void pthread_exit(void *value_ptr);
5
6 参数：
7     value_ptr:value_ptr不要指向一个局部变量。
8
```



```
9  返回值:
10     无返回值,跟进程一样,线程结束的时候无法返回到它的调用者(自身)
```

需要注意,pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的,不能在线程函数的栈上分配,因为当其它线程得到这个返回指针时线程函数已经退出了。

### pthread\_cancel函数

```
1  功能: 取消一个执行中的线程
2
3  原型:
4      int pthread_cancel(pthread_t thread);
5
6  参数:
7      thread:线程ID
8
9  返回值: 成功返回0; 失败返回错误码
```

## 3-4 线程等待

为什么需要线程等待?

- 已经退出的线程,其空间没有被释放,仍然在进程的地址空间内。
- 创建新的线程不会复用刚才退出线程的地址空间。

```
1  功能: 等待线程结束
2
3  原型
4      int pthread_join(pthread_t thread, void **value_ptr);
5
6  参数:
7      thread:线程ID
8      value_ptr:它指向一个指针,后者指向线程的返回值
9
10  返回值: 成功返回0; 失败返回错误码
```

调用该函数的线程将挂起等待,直到id为thread的线程终止。thread线程以不同的方法终止,通过pthread\_join得到的终止状态是不同的,总结如下:

1. 如果thread线程通过return返回,value\_ptr所指向的单元里存放的是thread线程函数的返回值。
2. 如果thread线程被别的线程调用pthread\_cancel异常终掉,value\_ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。

3. 如果thread线程是自己调用pthread\_exit终止的,value\_ptr所指向的单元存放的是传给pthread\_exit的参数。
4. 如果对thread线程的终止状态不感兴趣,可以传NULL给value\_ptr参数。

样例代码:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 void *thread1( void *arg )
8 {
9     printf("thread 1 returning ... \n");
10    int *p = (int*)malloc(sizeof(int));
11    *p = 1;
12    return (void*)p;
13 }
14
15 void *thread2( void *arg )
16 {
17     printf("thread 2 exiting ... \n");
18     int *p = (int*)malloc(sizeof(int));
19     *p = 2;
20     pthread_exit((void*)p);
21 }
22
23 void *thread3( void *arg )
24 {
25     while ( 1 ){ //
26         printf("thread 3 is running ... \n");
27         sleep(1);
28     }
29     return NULL;
30 }
31
32 int main( void )
33 {
34     pthread_t tid;
35     void *ret;
36
37     // thread 1 return
38     pthread_create(&tid, NULL, thread1, NULL);
39     pthread_join(tid, &ret);
40     printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
```

```

41     free(ret);
42
43     // thread 2 exit
44     pthread_create(&tid, NULL, thread2, NULL);
45     pthread_join(tid, &ret);
46     printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
47     free(ret);
48
49     // thread 3 cancel by other
50     pthread_create(&tid, NULL, thread3, NULL);
51     sleep(3);
52     pthread_cancel(tid);
53     pthread_join(tid, &ret);
54     if ( ret == PTHREAD_CANCELED )
55         printf("thread return, thread id %X, return code:PTHREAD_CANCELED\n",
tid);
56     else
57         printf("thread return, thread id %X, return code:NULL\n", tid);
58 }

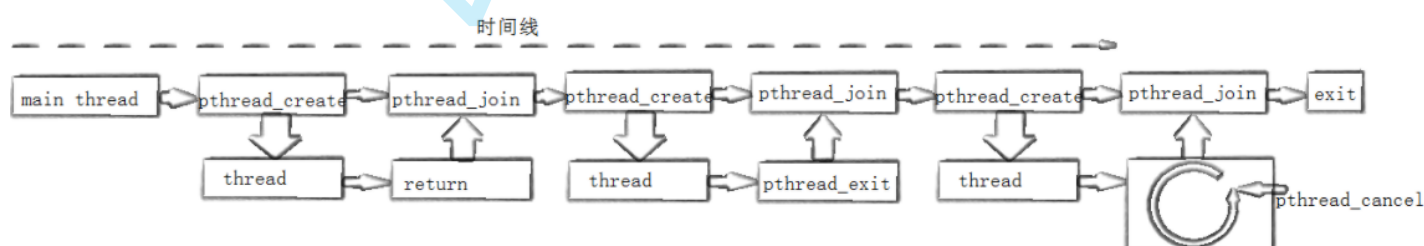
```

60 运行结果:

```

61     [root@localhost linux]# ./a.out
62     thread 1 returning ...
63     thread return, thread id 5AA79700, return code:1
64     thread 2 exiting ...
65     thread return, thread id 5AA79700, return code:2
66     thread 3 is running ...
67     thread 3 is running ...
68     thread 3 is running ...
69     thread return, thread id 5AA79700, return code:PTHREAD_CANCELED

```



### 3-5 分离线程

- 默认情况下，新创建的线程是joinable的，线程退出后，需要对其进行pthread\_join操作，否则无法释放资源，从而造成系统泄漏。
- 如果不关心线程的返回值，join是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。

```
1 int pthread_detach(pthread_t thread);
```

可以是线程组内其他线程对目标线程进行分离，也可以是线程自己分离：

```
1 pthread_detach(pthread_self());
```

joinable和分离是冲突的，一个线程不能既是joinable又是分离的。

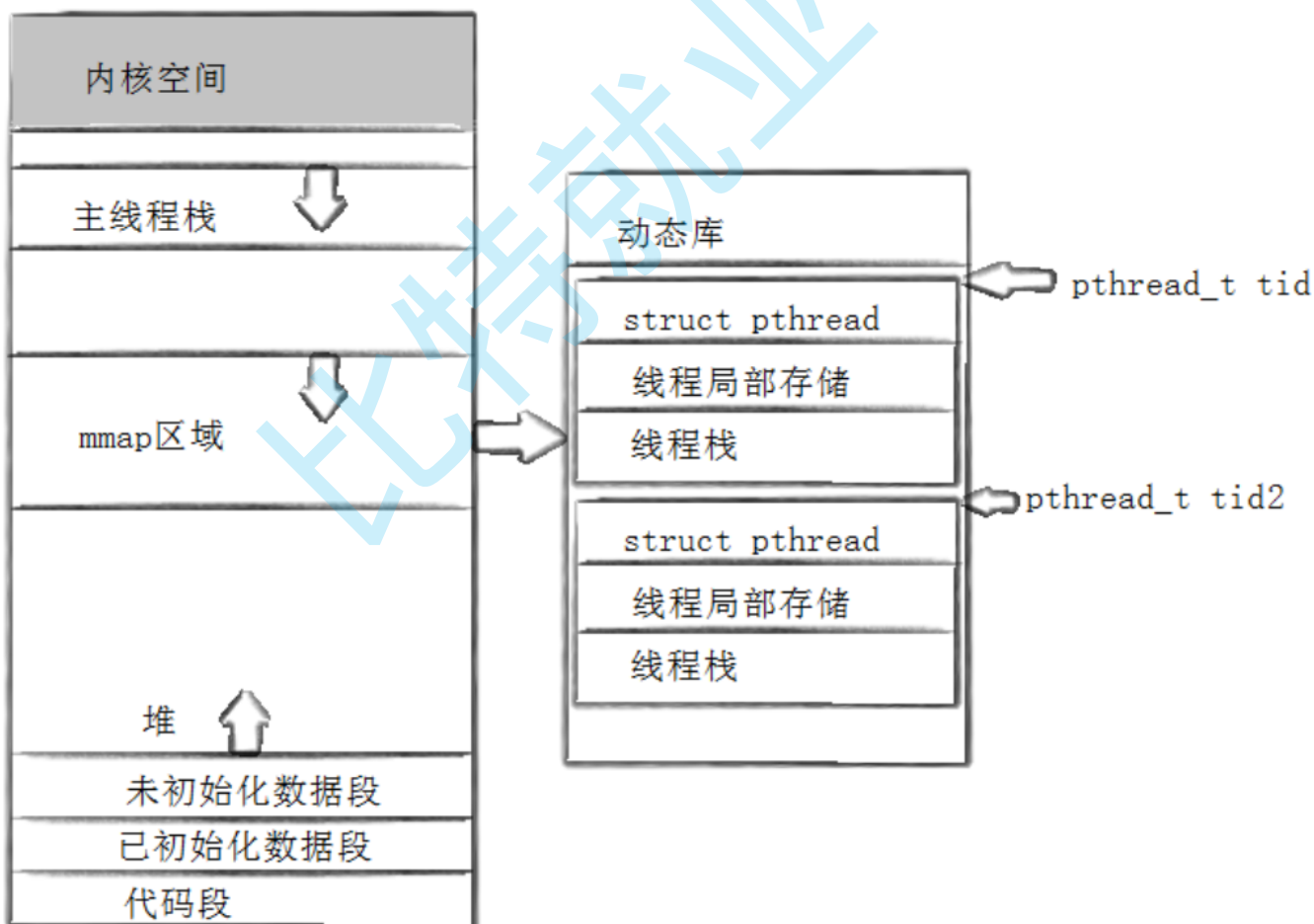
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 void *thread_run( void * arg )
8 {
9     pthread_detach(pthread_self());
10    printf("%s\n", (char*)arg);
11    return NULL;
12 }
13
14 int main( void )
15 {
16     pthread_t tid;
17     if ( pthread_create(&tid, NULL, thread_run, "thread1 run...") != 0 ) {
18         printf("create thread error\n");
19         return 1;
20     }
21
22     int ret = 0;
23     sleep(1); //很重要，要让线程先分离，再等待
24
25     if ( pthread_join(tid, NULL ) == 0 ) {
26         printf("pthread wait success\n");
27         ret = 0;
28     } else {
29         printf("pthread wait failed\n");
30         ret = 1;
31     }
32     return ret;
33 }
```

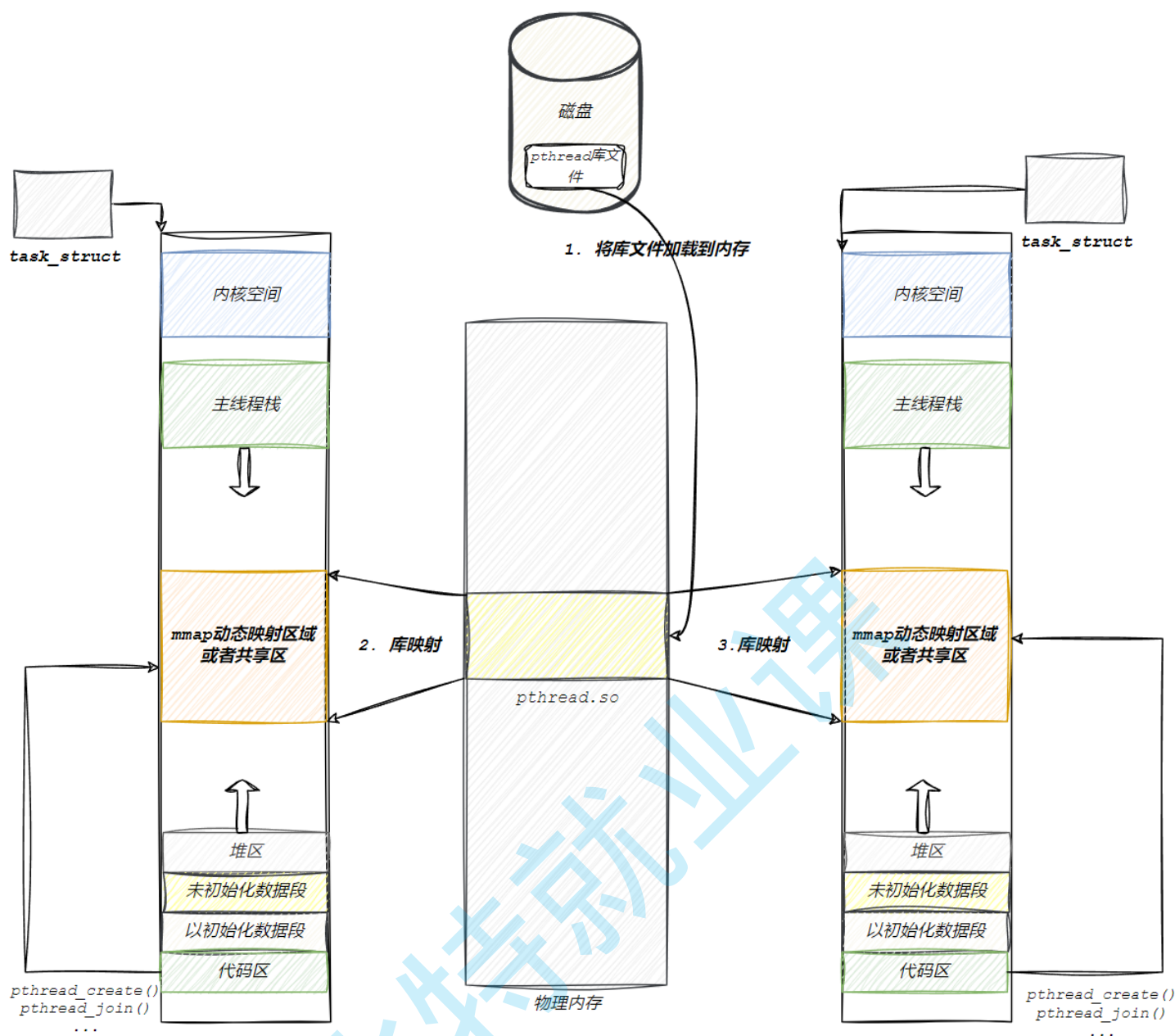
## 4. 线程ID及进程地址空间布局

- `pthread_create`函数会产生一个线程ID，存放在第一个参数指向的地址中。该线程ID和前面说的线程ID不是一回事。
- 前面讲的线程ID属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要有一个数值来唯一表示该线程。
- `pthread_create`函数第一个参数指向一个虚拟内存单元，该内存单元的地址即为新创建线程的线程ID，属于NPTL线程库的范畴。线程库的后续操作，就是根据该线程ID来操作线程的。
- 线程库NPTL提供了`pthread_self`函数，可以获得线程自身的ID：

```
1 pthread_t pthread_self(void);
```

`pthread_t` 到底是什么类型呢？取决于实现。对于Linux目前实现的NPTL实现而言，`pthread_t`类型的线程ID，本质就是一个进程地址空间上的一个地址。





## 5. 线程封装

```

1 // Thread.hpp
2 #pragma once
3
4 #include <iostream>
5 #include <string>
6 #include <functional>
7 #include <pthread.h>
8
9 namespace ThreadModule
10 {
11     // 原子计数器，方便形成线程名称
12     std::uint32_t cnt = 0;
13
14     // 线程要执行的外部方法，我们不考虑传参，后续有std::bind来进行类间耦合

```

```

15     using threadfunc_t = std::function<void()>;
16
17     // 线程状态
18     enum class TSTATUS
19     {
20         THREAD_NEW,
21         THREAD_RUNNING,
22         THREAD_STOP
23     };
24
25     // 线程
26     class Thread
27     {
28     private:
29         static void *run(void *obj)
30         {
31             Thread *self = static_cast<Thread *>(obj);
32             pthread_setname_np(pthread_self(), self->_name.c_str()); // 设置线程
名称
33             self->_status = TSTATUS::THREAD_RUNNING;
34             if (!self->_joined)
35             {
36                 pthread_detach(pthread_self());
37             }
38             self->_func();
39             return nullptr;
40         }
41         void SetName()
42         {
43             // 后期加锁保护
44             _name = "Thread-" + std::to_string(cnt++);
45         }
46
47     public:
48         Thread(threadfunc_t func) : _status(TSTATUS::THREAD_NEW),
49         _joined(true), _func(func)
50         {
51             SetName();
52         }
53         void EnableDetach()
54         {
55             if (_status == TSTATUS::THREAD_NEW)
56                 _joined = false;
57         }
58         void EnableJoined()
59         {
60             if (_status == TSTATUS::THREAD_NEW)

```



```

60         _joined = true;
61     }
62     bool Start()
63     {
64         if (_status == TSTATUS::THREAD_RUNNING)
65             return true;
66         int n = ::pthread_create(&_id, nullptr, run, this);
67         if (n != 0)
68             return false;
69         return true;
70     }
71     bool Join()
72     {
73         if (_joined)
74         {
75             int n = pthread_join(_id, nullptr);
76             if (n != 0)
77                 return false;
78             return true;
79         }
80         return false;
81     }
82     ~Thread() {}
83
84 private:
85     std::string _name;
86     pthread_t _id;
87     TSTATUS _status;
88     bool _joined;
89     threadfunc_t _func;
90 };
91 }
92

```

```

1  // main.cc
2
3  #include <iostream>
4  #include <unistd.h>
5  #include "test.hpp"
6
7  void hello1()
8  {
9      char buffer[64];
10     pthread_getname_np(pthread_self(), buffer, sizeof(buffer) - 1);
11     while (true)

```

```


12     {
13         std::cout << "hello world, " << buffer << std::endl;
14         sleep(1);
15     }
16 }
17 void hello2()
18 {
19     char buffer[64];
20     pthread_getname_np(pthread_self(), buffer, sizeof(buffer) - 1);
21     while (true)
22     {
23         std::cout << "hello world, " << buffer << std::endl;
24         sleep(1);
25     }
26 }
27
28 int main()
29 {
30     pthread_setname_np(pthread_self(), "main");
31     ThreadModule::Thread t1(hello1);
32     t1.Start();
33
34     ThreadModule::Thread t2(std::bind(&hello2));
35     t2.Start();
36
37     t1.Join();
38     t2.Join();
39
40     return 0;
41 }

```

```

1 // 运行结果查询
2 $ ps -aL
3      PID      LWP TTY          TIME CMD
4  195828  195828 pts/1      00:00:00 main
5  195828  195829 pts/1      00:00:00 Thread-0
6  195828  195830 pts/1      00:00:00 Thread-1
7

```

 如果要像C++11那样进行可变参数的传递，是可以这样设计的，但是太麻烦了，真到了哪一步，就直接用c++11吧，我们的目标主要是理解系统概念对象化，此处不做复杂设计，而且后续可以使用std::bind来进行对象间调用

## 6. 附录

### 6-1 源码阅读，理解线程

以下是glibc-2.4中pthread源码相关内容:

路径: `nptl/pthread_create.c`

```
1 int __pthread_create_2_1(newthread, attr, start_routine, arg)
2 pthread_t *newthread;
3 const pthread_attr_t *attr;
4 void *(*start_routine)(void *);
5 void *arg;
6 {
7     STACK_VARIABLES;
8     // 重点1: 线程属性, 虽然我们不设置, 但是不妨碍我们了解
9     const struct pthread_attr *iattr = (struct pthread_attr *)attr;
10    if (iattr == NULL)
11        /* Is this the best idea? On NUMA machines this could mean
12         accessing far-away memory. */
13        iattr = &default_attr;
14
15    // 重点2: 传说中的原生线程库中的用来描述线程的tcb
16    struct pthread *pd = NULL;
17    // 重点3: ALLOCATE_STACK会在先申请struct pthread对象, 当然其实是申请一大块空间,
18    // struct pthread在空间的开头, 一会追
19    int err = ALLOCATE_STACK(iattr, &pd);
20    if (__builtin_expect(err != 0, 0))
21        /* Something went wrong. Maybe a parameter of the attributes is
22         invalid or we could not allocate memory. */
23        versioned_symbol return err;
24
25    /* Initialize the TCB. All initializations with zero should be
26     performed in 'get_cached_stack'. This way we avoid doing this if
27     the stack freshly allocated with 'mmap'. */
28
29    #ifdef TLS_TCB_AT_TP
30        /* Reference to the TCB itself. */
31        pd->header.self = pd;
32
33        /* Self-reference for TLS. */
34        pd->header.tcb = pd;
35    #endif
36
37    /* Store the address of the start routine and the parameter. Since
38     we do not start the function directly the stillborn thread will
```

```

39     get the information from its thread descriptor. */
40     // 重点4: 向线程tcb中设置未来要执行的方法的地址和参数
41     pd->start_routine = start_routine;
42     pd->arg = arg;
43
44     /* Copy the thread attribute flags. */
45     struct pthread *self = THREAD_SELF;
46     pd->flags = ((iattr->flags & ~(ATTR_FLAG_SCHED_SET | ATTR_FLAG_POLICY_SET))
47 | (self->flags & (ATTR_FLAG_SCHED_SET | ATTR_FLAG_POLICY_SET)));
48
49     /* Initialize the field for the ID of the thread which is waiting
50        for us. This is a self-reference in case the thread is created
51        detached. */
52     pd->joinid = iattr->flags & ATTR_FLAG_DETACHSTATE ? pd : NULL;
53
54     /* The debug events are inherited from the parent. */
55     pd->eventbuf = self->eventbuf;
56
57     /* Copy the parent's scheduling parameters. The flags will say what
58        is valid and what is not. */
59     pd->schedpolicy = self->schedpolicy;
60     pd->schedparam = self->schedparam;
61
62     /* Copy the stack guard canary. */
63     #ifdef THREAD_COPY_STACK_GUARD
64     THREAD_COPY_STACK_GUARD(pd);
65     #endif
66
67     /* Copy the pointer guard value. */
68     #ifdef THREAD_COPY_POINTER_GUARD
69     THREAD_COPY_POINTER_GUARD(pd);
70     #endif
71     // 一堆参数设定, 我们不关心
72     /* Determine scheduling parameters for the thread. */
73     if (attr != NULL && __builtin_expect((iattr->flags &
74 ATTR_FLAG_NOTINHERITSCHED) != 0, 0) && (iattr->flags & (ATTR_FLAG_SCHED_SET |
75 ATTR_FLAG_POLICY_SET)) != 0)
76     {
77         INTERNAL_SYSCALL_DECL(scerr);
78
79         /* Use the scheduling parameters the user provided. */
80         if (iattr->flags & ATTR_FLAG_POLICY_SET)
81             pd->schedpolicy = iattr->schedpolicy;
82         else if ((pd->flags & ATTR_FLAG_POLICY_SET) == 0)
83         {
84             pd->schedpolicy = INTERNAL_SYSCALL(sched_getscheduler, scerr, 1, 0);
85             pd->flags |= ATTR_FLAG_POLICY_SET;
86         }
87     }

```

```

83     }
84
85     if (iattr->flags & ATTR_FLAG_SCHED_SET)
86         memcpy(&pd->schedparam, &iattr->schedparam,
87             sizeof(struct sched_param));
88     else if ((pd->flags & ATTR_FLAG_SCHED_SET) == 0)
89     {
90         INTERNAL_SYSCALL(sched_getparam, scerr, 2, 0, &pd->schedparam);
91         pd->flags |= ATTR_FLAG_SCHED_SET;
92     }
93
94     /* Check for valid priorities. */
95     int minprio = INTERNAL_SYSCALL(sched_get_priority_min, scerr, 1,
96         iattr->schedpolicy);
97     int maxprio = INTERNAL_SYSCALL(sched_get_priority_max, scerr, 1,
98         iattr->schedpolicy);
99     if (pd->schedparam.sched_priority < minprio || pd-
100 >schedparam.sched_priority > maxprio)
101     {
102         err = EINVAL;
103         goto errout;
104     }
105
106     /* Pass the descriptor to the caller. */
107     // 重点5: 把pd (就是线程控制块地址) 作为ID, 传递出去, 所以上层拿到的就是一个虚拟地址
108     *newthread = (pthread_t)pd;
109
110     /* Remember whether the thread is detached or not. In case of an
111        error we have to free the stacks of non-detached stillborn
112        threads. */
113
114     // 重点6: 检测线程属性是否分离, 这个很好理解
115     bool is_detached = IS_DETACHED(pd);
116
117     /* Start the thread. */
118     err = create_thread(pd, iattr, STACK_VARIABLES_ARGS); // 重点函数
119     if (err != 0)
120     {
121         /* Something went wrong. Free the resources. */
122         if (!is_detached)
123         {
124             errout:
125             __deallocate_stack(pd);
126         }
127         return err;
128     }

```

```

129
130     return 0;
131 }
132 // 版本确认信息，意思就是如果用的库是GLIBC_2_1，pthread_create函数就是
    __pthread_create_2_1
133 versioned_symbol(libpthread, __pthread_create_2_1, pthread_create, GLIBC_2_1);

```

线程属性：

```

1 struct pthread_attr
2 {
3     /* Scheduler parameters and priority. */
4     struct sched_param schedparam;
5     int schedpolicy;
6     /* Various flags like detachstate, scope, etc. */
7     int flags;
8     /* Size of guard area. */
9     size_t guardsize;
10    /* Stack handling. */
11    void *stackaddr;
12    size_t stacksize;
13    /* Affinity map. */
14    cpu_set_t *cpuset;
15    size_t cpusetsize;
16 };

```

线程tcb:

```

1 /* Thread descriptor data structure. */
2 struct pthread
3 {
4     union
5     {
6         #if !TLS_DTV_AT_TP
7             /* This overlaps the TCB as used for TLS without threads (see tls.h). */
8             tcbhead_t header;
9         #else
10            struct
11            {
12                int multiple_threads;
13            } header;
14        #endif
15    };

```

```

16     /* This extra padding has no special purpose, and this structure layout
17        is private and subject to change without affecting the official ABI.
18        We just have it here in case it might be convenient for some
19        implementation-specific instrumentation hack or suchlike. */
20     void *__padding[16];
21 };
22
23     /* This descriptor's link on the 'stack_used' or '__stack_user' list. */
24     list_t list;
25
26     /* Thread ID - which is also a 'is this thread descriptor (and
27        therefore stack) used' flag. */
28     pid_t tid;
29
30     /* Process ID - thread group ID in kernel speak. */
31     pid_t pid;
32
33     /* List of robust mutexes the thread is holding. */
34 #ifdef __PTHREAD_MUTEX_HAVE_PREV
35     __pthread_list_t robust_list;
36
37 # define ENQUEUE_MUTEX(mutex) \
38     do { \
39         __pthread_list_t *next = THREAD_GETMEM (THREAD_SELF, robust_list.__next); \
40         next->__prev = &mutex->__data.__list; \
41         mutex->__data.__list.__next = next; \
42         mutex->__data.__list.__prev = &THREAD_SELF->robust_list; \
43         THREAD_SETMEM (THREAD_SELF, robust_list.__next, &mutex->__data.__list); \
44     } while (0)
45 # define DEQUEUE_MUTEX(mutex) \
46     do { \
47         mutex->__data.__list.__next->__prev = mutex->__data.__list.__prev; \
48         mutex->__data.__list.__prev->__next = mutex->__data.__list.__next; \
49         mutex->__data.__list.__prev = NULL; \
50         mutex->__data.__list.__next = NULL; \
51     } while (0)
52 #else
53     __pthread_slist_t robust_list;
54
55 # define ENQUEUE_MUTEX(mutex) \
56     do { \
57         mutex->__data.__list.__next \
58         = THREAD_GETMEM (THREAD_SELF, robust_list.__next); \
59         THREAD_SETMEM (THREAD_SELF, robust_list.__next, &mutex->__data.__list); \
60     } while (0)
61 # define DEQUEUE_MUTEX(mutex) \
62     do { \

```



```

63     __pthread_slist_t *runp = THREAD_GETMEM (THREAD_SELF, robust_list.__next);\
64     if (runp == &mutex->__data.__list) \
65         THREAD_SETMEM (THREAD_SELF, robust_list.__next, runp->__next); \
66     else \
67     { \
68 while (runp->__next != &mutex->__data.__list) \
69     runp = runp->__next; \
70 \
71     runp->__next = runp->__next->__next; \
72     mutex->__data.__list.__next = NULL; \
73     } \
74 } while (0)
75 #endif
76
77 /* List of cleanup buffers. */
78 struct _pthread_cleanup_buffer *cleanup;
79
80 /* Unwind information. */
81 struct pthread_unwind_buf *cleanup_jmp_buf;
82 #define HAVE_CLEANUP_JMP_BUF
83
84 /* Flags determining processing of cancellation. */
85 int cancelhandling;
86 /* Bit set if cancellation is disabled. */
87 #define CANCELSTATE_BIT 0
88 #define CANCELSTATE_BITMASK 0x01
89 /* Bit set if asynchronous cancellation mode is selected. */
90 #define CANCELTYPE_BIT 1
91 #define CANCELTYPE_BITMASK 0x02
92 /* Bit set if canceling has been initiated. */
93 #define CANCELING_BIT 2
94 #define CANCELING_BITMASK 0x04
95 /* Bit set if canceled. */
96 #define CANCELED_BIT 3
97 #define CANCELED_BITMASK 0x08
98 /* Bit set if thread is exiting. */
99 #define EXITING_BIT 4
100 #define EXITING_BITMASK 0x10
101 /* Bit set if thread terminated and TCB is freed. */
102 #define TERMINATED_BIT 5
103 #define TERMINATED_BITMASK 0x20
104 /* Bit set if thread is supposed to change XID. */
105 #define SETXID_BIT 6
106 #define SETXID_BITMASK 0x40
107 /* Mask for the rest. Helps the compiler to optimize. */
108 #define CANCEL_RESTMASK 0xffffffff80
109

```

```

110 #define CANCEL_ENABLED_AND_CANCELED(value) \
111     (((value) & (CANCELSTATE_BITMASK | CANCELED_BITMASK | EXITING_BITMASK \
112         | CANCEL_RESTMASK | TERMINATED_BITMASK)) == CANCELED_BITMASK)
113 #define CANCEL_ENABLED_AND_CANCELED_AND_ASYNCHRONOUS(value) \
114     (((value) & (CANCELSTATE_BITMASK | CANCELTYPE_BITMASK | CANCELED_BITMASK \
115         | EXITING_BITMASK | CANCEL_RESTMASK | TERMINATED_BITMASK)) \
116     == (CANCELTYPE_BITMASK | CANCELED_BITMASK))
117
118 /* We allocate one block of references here. This should be enough
119 to avoid allocating any memory dynamically for most applications. */
120 struct pthread_key_data
121 {
122     /* Sequence number. We use uintptr_t to not require padding on
123     32- and 64-bit machines. On 64-bit machines it helps to avoid
124     wrapping, too. */
125     uintptr_t seq;
126
127     /* Data pointer. */
128     void *data;
129 } specific_1stblock[PTHREAD_KEY_2NDLEVEL_SIZE];
130
131 /* Two-level array for the thread-specific data. */
132 struct pthread_key_data *specific[PTHREAD_KEY_1STLEVEL_SIZE];
133
134 /* Flag which is set when specific data is set. */
135 bool specific_used;
136
137 /* True if events must be reported. */
138 bool report_events;
139
140 /* True if the user provided the stack. */
141 bool user_stack;
142
143 /* True if thread must stop at startup time. */
144 bool stopped_start;
145
146 /* Lock to synchronize access to the descriptor. */
147 lll_lock_t lock;
148
149 /* Lock for synchronizing setxid calls. */
150 lll_lock_t setxid_futex;
151
152 #if HP_TIMING_AVAIL
153     /* Offset of the CPU clock at start thread start time. */
154     hp_timing_t cpuclock_offset;
155 #endif
156

```

```

157  /* If the thread waits to join another one the ID of the latter is
158     stored here.
159
160     In case a thread is detached this field contains a pointer of the
161     TCB if the thread itself. This is something which cannot happen
162     in normal operation. */
163  struct pthread *joinid;
164  /* Check whether a thread is detached. */
165  #define IS_DETACHED(pd) ((pd)->joinid == (pd))
166
167  /* Flags. Including those copied from the thread attribute. */
168  int flags;
169
170  /* The result of the thread function. */
171  // 线程运行完毕，返回值就是void*，最后的返回值就放在tcb中的该变量里面
172  // 所以我们用pthread_join获取线程退出信息的时候，就是读取该结构体
173  // 另外，要能理解线程执行流可以退出，但是tcb可以暂时保留，这句话
174  void *result;
175
176  /* Scheduling parameters for the new thread. */
177  struct sched_param schedparam;
178  int schedpolicy;
179
180  /* Start position of the code to be executed and the argument passed
181     to the function. */
182  // 用户指定的方法和参数
183  void *(*start_routine) (void *);
184  void *arg;
185
186  /* Debug state. */
187  td_eventbuf_t eventbuf;
188  /* Next descriptor with a pending event. */
189  struct pthread *nextevent;
190
191  #ifdef HAVE_FORCED_UNWIND
192  /* Machine-specific unwind info. */
193  struct _Unwind_Exception exc;
194  #endif
195
196  /* If nonzero pointer to area allocated for the stack and its
197     size. */
198  // 线程自己的栈和大小
199  void *stackblock;
200  size_t stackblock_size;
201  /* Size of the included guard area. */
202  size_t guardsize;
203  /* This is what the user specified and what we will report. */

```

```

204     size_t reported_guardsize;
205
206     /* Resolver state. */
207     struct __res_state res;
208
209     /* This member must be last. */
210     char end_padding[];
211
212     #define PTHREAD_STRUCT_END_PADDING \
213         (sizeof (struct pthread) - offsetof (struct pthread, end_padding))
214 } __attribute ((aligned (TCB_ALIGNMENT)));
215

```

## create\_thread

```

1  static int
2  create_thread(struct pthread *pd, const struct pthread_attr *attr,
3               STACK_VARIABLES_PARMS)
4  {
5      #ifdef TLS_TCB_AT_TP
6          assert(pd->header.tcb != NULL);
7      #endif
8
9      /* We rely heavily on various flags the CLONE function understands:
10
11         CLONE_VM, CLONE_FS, CLONE_FILES
12         These flags select semantics with shared address space and
13         file descriptors according to what POSIX requires.
14
15         CLONE_SIGNAL
16         This flag selects the POSIX signal semantics.
17
18         CLONE_SETTLS
19         The sixth parameter to CLONE determines the TLS area for the
20         new thread.
21
22         CLONE_PARENT_SETTID
23         The kernel writes the thread ID of the newly created thread
24         into the location pointed to by the fifth parameter to CLONE.
25
26         Note that it would be semantically equivalent to use
27         CLONE_CHILD_SETTID but it is more expensive in the kernel.
28
29         CLONE_CHILD_CLEARTID
30         The kernel clears the thread ID of a thread that has called


```

```

31  sys_exit() in the location pointed to by the seventh parameter
32  to CLONE.
33
34  CLONE_DETACHED
35  No signal is generated if the thread exists and it is
36  automatically reaped.
37
38  The termination signal is chosen to be zero which means no signal
39  is sent.  */
40  int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGNAL |
    CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
41  #if __ASSUME_NO_CLONE_DETACHED == 0
42      | CLONE_DETACHED
43  #endif
44      | 0);
45
46  if (__builtin_expect(THREAD_GETMEM(THREAD_SELF, report_events), 0))
47  {
48      /* The parent thread is supposed to report events. Check whether
49      the TD_CREATE event is needed, too.  */
50      const int _idx = __td_eventword(TD_CREATE);
51      const uint32_t _mask = __td_eventmask(TD_CREATE);
52
53      if ((_mask & (__nptl_threads_events.event_bits[_idx] | pd-
    >eventbuf.eventmask.event_bits[_idx])) != 0)
54      {
55          /* We always must have the thread start stopped.  */
56          pd->stopped_start = true;
57
58          /* Create the thread. We always create the thread stopped
59          so that it does not get far before we tell the debugger.  */
60          int res = do_clone(pd, attr, clone_flags, start_thread,
61                          STACK_VARIABLES_ARGS, 1);
62          if (res == 0)
63          {
64              /* Now fill in the information about the new thread in
65              the newly created thread's data structure. We cannot let
66              the new thread do this since we don't know whether it was
67              already scheduled when we send the event.  */
68              pd->eventbuf.eventnum = TD_CREATE;
69              pd->eventbuf.eventdata = pd;
70
71              /* Enqueue the descriptor.  */
72              do
73                  pd->nextevent = __nptl_last_event;
74              while (atomic_compare_and_exchange_bool_acq(&__nptl_last_event,
75                  pd, pd->nextevent) != 0);

```

```

76
77     /* Now call the function which signals the event. */
78     __nptl_create_event();
79
80     /* And finally restart the new thread. */
81     lll_unlock(pd->lock);
82 }
83
84     return res;
85 }
86 }
87
88 #ifdef NEED_DL_SYSINFO
89     assert(THREAD_SELF_SYSINFO == THREAD_SYSINFO(pd));
90 #endif
91
92     /* Determine whether the newly created threads has to be started
93     stopped since we have to set the scheduling parameters or set the
94     affinity. */
95     bool stopped = false;
96     if (attr != NULL && (attr->cpuset != NULL || (attr->flags &
97 ATTR_FLAG_NOTINHERITSCHED) != 0))
98         stopped = true;
99     pd->stopped_start = stopped;
100
101     /* Actually create the thread. */
102     int res = do_clone(pd, attr, clone_flags, start_thread,
103                     STACK_VARIABLES_ARGS, stopped);
104
105     if (res == 0 && stopped)
106         /* And finally restart the new thread. */
107         lll_unlock(pd->lock);
108     return res;
109 }
110

```

## do\_clone

```

1 static int
2 do_clone(struct pthread *pd, const struct pthread_attr *attr,
3         int clone_flags, int (*fct)(void *), STACK_VARIABLES_PARMS,
4         int stopped)
5 {
6     #ifdef PREPARE_CREATE

```

```

7   PREPARE_CREATE;
8   #endif
9
10  if (stopped)
11      /* We Make sure the thread does not run far by forcing it to get a
12         lock. We lock it here too so that the new thread cannot continue
13         until we tell it to. */
14      lll_lock(pd->lock);
15
16      /* One more thread. We cannot have the thread do this itself, since it
17         might exist but not have been scheduled yet by the time we've returned
18         and need to check the value to behave correctly. We must do it before
19         creating the thread, in case it does get scheduled first and then
20         might mistakenly think it was the only thread. In the failure case,
21         we momentarily store a false value; this doesn't matter because there
22         is no kosher thing a signal handler interrupting us right here can do
23         that cares whether the thread count is correct. */
24      atomic_increment(&__nptl_nthreads);
25      // 执行特性体系结构下的clone函数
26      if (ARCH_CLONE(fct, STACK_VARIABLES_ARGS, clone_flags,
27                    pd, &pd->tid, TLS_VALUE, &pd->tid) == -1)
28      {
29          atomic_decrement(&__nptl_nthreads); /* Oops, we lied for a second. */
30
31          /* Failed. If the thread is detached, remove the TCB here since
32             the caller cannot do this. The caller remembered the thread
33             as detached and cannot reverify that it is not since it must
34             not access the thread descriptor again. */
35          if (IS_DETACHED(pd))
36              __deallocate_stack(pd);
37
38          return errno;
39      }
40
41      /* Now we have the possibility to set scheduling parameters etc. */
42      // 下面是调用相关系统调用, 设置轻量级进程的调度参数和一些异常处理, 不关心
43      if (__builtin_expect(stopped != 0, 0))
44      {
45          INTERNAL_SYSCALL_DECL(err);
46          int res = 0;
47
48          /* Set the affinity mask if necessary. */
49          if (attr->cpuset != NULL)
50          {
51              res = INTERNAL_SYSCALL(sched_setaffinity, err, 3, pd->tid,
52                                    sizeof(cpu_set_t), attr->cpuset);
53          }
54      }
55  }
56  }
57  }
58  }
59  }
60  }
61  }
62  }
63  }
64  }
65  }
66  }
67  }
68  }
69  }
70  }
71  }
72  }
73  }
74  }
75  }
76  }
77  }
78  }
79  }
80  }
81  }
82  }
83  }
84  }
85  }
86  }
87  }
88  }
89  }
90  }
91  }
92  }
93  }
94  }
95  }
96  }
97  }
98  }
99  }
100 }
```



```

54     if (__builtin_expect(INTERNAL_SYSCALL_ERROR_P(res, err), 0))
55     {
56         /* The operation failed. We have to kill the thread. First
57         send it the cancellation signal. */
58         INTERNAL_SYSCALL_DECL(err2);
59         err_out:
60         #if __ASSUME_TGKILL
61             (void)INTERNAL_SYSCALL(tgkill, err2, 3,
62                                     THREAD_GETMEM(THREAD_SELF, pid),
63                                     pd->tid, SIGCANCEL);
64         #else
65             (void)INTERNAL_SYSCALL(tkill, err2, 2, pd->tid, SIGCANCEL);
66         #endif
67
68         return (INTERNAL_SYSCALL_ERROR_P(res, err)
69                 ? INTERNAL_SYSCALL_ERRNO(res, err)
70                 : 0);
71     }
72 }
73
74 /* Set the scheduling parameters. */
75 if ((attr->flags & ATTR_FLAG_NOTINHERITSCHED) != 0)
76 {
77     res = INTERNAL_SYSCALL(sched_setscheduler, err, 3, pd->tid,
78                             pd->schedpolicy, &pd->schedparam);
79
80     if (__builtin_expect(INTERNAL_SYSCALL_ERROR_P(res, err), 0))
81         goto err_out;
82 }
83 }
84
85 /* We now have for sure more than one thread. The main thread might
86 not yet have the flag set. No need to set the global variable
87 again if this is what we use. */
88 THREAD_SETMEM(THREAD_SELF, header.multiple_threads, 1);
89
90 return 0;
91 }

```

```

1 #define ARCH_CLONE __clone

```

```

2

```

```

3

```

4 \_\_clone是glibc用汇编封装的一个调用clone系统调用的函数，所以

5 \_\_clone的实现就是汇编，贴一份代码（sysdeps/unix/sysv/linux/x86\_64）：

```

6 ENTRY (BP_SYM (__clone))

```

```

7      /* Sanity check arguments. */
8      movq    $-EINVAL,%rax
9      testq   %rdi,%rdi                /* no NULL function pointers */
10     jz      SYSCALL_ERROR_LABEL
11     testq   %rsi,%rsi                /* no NULL stack pointers */
12     jz      SYSCALL_ERROR_LABEL
13
14     /* Insert the argument onto the new stack. */
15     subq     $16,%rsi
16     movq     %rcx,8(%rsi)
17
18     /* Save the function pointer. It will be popped off in the
19        child in the ebx frobbing below. */
20     movq     %rdi,0(%rsi)
21
22     /* Do the system call. */
23     movq     %rdx, %rdi
24     movq     %r8, %rdx
25     movq     %r9, %r8
26     movq     8(%rsp), %r10
27     movl     $SYS_ify(clone),%eax // 获取系统调用号
28
29     /* End FDE now, because in the child the unwind info will be
30        wrong. */
31     cfi_endproc;
32     syscall // 陷入内核(x86_32是int 80), 要求内核创建轻量级进程
33
34     testq    %rax,%rax
35     jl      SYSCALL_ERROR_LABEL
36     jz      L(thread_start)
37
38 这部分代码了解即可。
39

```

下面我们追一下空间申请：`int err = ALLOCATE_STACK(iattr, &pd);`

```

1  源码路径: nptl/allocatestack.c
2  //空间申请的函数, 其实就是一个宏
3  #define ALLOCATE_STACK(attr, pd) \
4      allocate_stack(attr, pd, &stackaddr, &stacksize)
5
6
7  static int
8  allocate_stack(const struct pthread_attr *attr, struct pthread **pdp,
9                ALLOCATE_STACK_PARMS)

```

```

10 {
11     struct pthread *pd;
12     size_t size;
13     size_t pagesize_m1 = __getpagesize() - 1;
14     void *stacktop;
15
16     assert(attr != NULL);
17     assert(powerof2(pagesize_m1 + 1));
18     assert(TCB_ALIGNMENT >= STACK_ALIGN);
19
20     /* Get the stack size from the attribute if it is set. Otherwise we
21        use the default we determined at start time. */
22     size = attr->stacksize ? : __default_stacksize; // 获取栈大小, 用户没设置就默认
23
24     /* Get memory for the stack. */
25     // 如果已经用户已经在线程属性里面设置了空间, 就直接用, 我们是默认, 这部分代码直接不看
26     if (__builtin_expect(attr->flags & ATTR_FLAG_STACKADDR, 0))
27     {
28         uintptr_t adj;
29
30         /* If the user also specified the size of the stack make sure it
31            is large enough. */
32         if (attr->stacksize != 0 && attr->stacksize < (__static_tls_size +
33            MINIMAL_REST_STACK))
34             return EINVAL;
35
36         /* Adjust stack size for alignment of the TLS block. */
37         #if TLS_TCB_AT_TP
38             adj = ((uintptr_t)attr->stackaddr - TLS_TCB_SIZE) & __static_tls_align_m1;
39             assert(size > adj + TLS_TCB_SIZE);
40         #elif TLS_DTV_AT_TP
41             adj = ((uintptr_t)attr->stackaddr - __static_tls_size) &
42                 __static_tls_align_m1;
43             assert(size > adj);
44         #endif
45
46         /* The user provided some memory. Let's hope it matches the
47            size... We do not allocate guard pages if the user provided
48            the stack. It is the user's responsibility to do this if it
49            is wanted. */
50         #if TLS_TCB_AT_TP
51             pd = (struct pthread *)((uintptr_t)attr->stackaddr - TLS_TCB_SIZE - adj);
52         #elif TLS_DTV_AT_TP
53             pd = (struct pthread *)(((uintptr_t)attr->stackaddr - __static_tls_size -
54                 adj) - TLS_PRE_TCB_SIZE);
55         #endif
56     }
57 }

```

```

54  /* The user provided stack memory needs to be cleared. */
55  memset(pd, '\0', sizeof(struct pthread));
56
57  /* The first TSD block is included in the TCB. */
58  pd->specific[0] = pd->specific_1stblock;
59
60  /* Remember the stack-related values. */
61  pd->stackblock = (char *)attr->stackaddr - size;
62  pd->stackblock_size = size;
63
64  /* This is a user-provided stack. It will not be queued in the
65  stack cache nor will the memory (except the TLS memory) be freed. */
66  pd->user_stack = true;
67
68  /* This is at least the second thread. */
69  pd->header.multiple_threads = 1;
70  #ifndef TLS_MULTIPLE_THREADS_IN_TCB
71  __pthread_multiple_threads = *__libc_multiple_threads_ptr = 1;
72  #endif
73
74  #ifdef NEED_DL_SYSINFO
75  /* Copy the sysinfo value from the parent. */
76  THREAD_SYSINFO(pd) = THREAD_SELF_SYSINFO;
77  #endif
78
79  /* The process ID is also the same as that of the caller. */
80  pd->pid = THREAD_GETMEM(THREAD_SELF, pid);
81
82  /* List of robust mutexes. */
83  #ifdef __PTHREAD_MUTEX_HAVE_PREV
84  pd->robust_list.__prev = &pd->robust_list;
85  #endif
86  pd->robust_list.__next = &pd->robust_list;
87
88  /* Allocate the DTV for this thread. */
89  if (_dl_allocate_tls(TLS_TPADJ(pd)) == NULL)
90  {
91      /* Something went wrong. */
92      assert(errno == ENOMEM);
93      return EAGAIN;
94  }
95
96  /* Prepare to modify global data. */
97  lll_lock(stack_cache_lock);
98
99  /* And add to the list of stacks in use. */
100 list_add(&pd->list, &__stack_user);

```

```

101
102     lll_unlock(stack_cache_lock);
103 }
104 else
105 {
106     // 下面的都是库内部自己做的，我们关心的
107     /* Allocate some anonymous memory. If possible use the cache. */
108     size_t guardsize;
109     size_t reqsize;
110     void *mem;
111     const int prot = (PROT_READ | PROT_WRITE | ((GL(dl_stack_flags) & PF_X) ?
112 PROT_EXEC : 0));
113
114 #if COLORING_INCREMENT != 0
115     /* Add one more page for stack coloring. Don't do it for stacks
116 with 16 times pagesize or larger. This might just cause
117 unnecessary misalignment. */
118     if (size <= 16 * pagesize_m1)
119         size += pagesize_m1 + 1;
120 #endif
121
122     /* Adjust the stack size for alignment. */
123     size &= ~__static_tls_align_m1; // 设置空间对齐
124     assert(size != 0);
125
126     /* Make sure the size of the stack is enough for the guard and
127 eventually the thread descriptor. */
128     guardsize = (attr->guardsize + pagesize_m1) & ~pagesize_m1;
129     if (__builtin_expect(size < ((guardsize + __static_tls_size +
130 MINIMAL_REST_STACK + pagesize_m1) & ~pagesize_m1),
131 0))
132     /* The stack is too small (or the guard too large). */
133     return EINVAL;
134
135     /* Try to get a stack from the cache. */
136     // 先尝试从pthread缓存中申请空间
137     reqsize = size;
138     pd = get_cached_stack(&size, &mem);
139     if (pd == NULL)
140     {
141         /* To avoid aliasing effects on a larger scale than pages we
142 adjust the allocated stack size if necessary. This way
143 allocations directly following each other will not have
144 aliasing problems. */
145 #if MULTI_PAGE_ALIASING != 0
146         if ((size % MULTI_PAGE_ALIASING) == 0)
147             size += pagesize_m1 + 1;
148

```

```

146 #endif
147 // 缓存申请失败，就在堆空间申请私有的匿名内存空间，这里mmap类似malloc
148 // 当然他也可以作为共享内存的实现，类似原理我们接触过，这个功能和当前无关
149 mem = mmap(NULL, size, prot,
150             MAP_PRIVATE | MAP_ANONYMOUS | ARCH_MAP_FLAGS, -1, 0);
151
152 if (__builtin_expect(mem == MAP_FAILED, 0))
153 {
154 #ifdef ARCH_RETRY_MMAP
155     mem = ARCH_RETRY_MMAP(size);
156     if (__builtin_expect(mem == MAP_FAILED, 0))
157 #endif
158         return errno;
159 }
160
161 /* SIZE is guaranteed to be greater than zero.
162    So we can never get a null pointer back from mmap. */
163 assert(mem != NULL);
164
165 #if COLORING_INCREMENT != 0
166     /* Atomically increment NCREATED. */
167     unsigned int ncreated = atomic_increment_val(&nptl_ncreated);
168
169     /* We chose the offset for coloring by incrementing it for
170        every new thread by a fixed amount. The offset used
171        module the page size. Even if coloring would be better
172        relative to higher alignment values it makes no sense to
173        do it since the mmap() interface does not allow us to
174        specify any alignment for the returned memory block. */
175     size_t coloring = (ncreated * COLORING_INCREMENT) & pagesize_m1;
176
177     /* Make sure the coloring offsets does not disturb the alignment
178        of the TCB and static TLS block. */
179     if (__builtin_expect((coloring & __static_tls_align_m1) != 0, 0))
180         coloring = (((coloring + __static_tls_align_m1) & ~
181                     (__static_tls_align_m1)) & ~pagesize_m1);
182 #else
183     /* Unless specified we do not make any adjustments. */
184 #define coloring 0
185 #endif
186
187 /* Place the thread descriptor at the end of the stack. */
188 // 下面的代码其实就是我们课件中的图，这里是在申请的空间中确定struct thread(tcb)
// 的地址
189 #if TLS_TCB_AT_TP
190     pd = (struct pthread *)((char *)mem + size - coloring) - 1;
191 #elif TLS_DTV_AT_TP

```

```

191     pd = (struct pthread *)((((uintptr_t)mem + size - coloring -
    __static_tls_size) & ~__static_tls_align_m1) - TLS_PRE_TCB_SIZE);
192 #endif
193
194     /* Remember the stack-related values. */
195     // 记录下来整个空间的地址和大小
196     pd->stackblock = mem;
197     pd->stackblock_size = size;
198
199     /* We allocated the first block thread-specific data array.
    This address will not change for the lifetime of this
    descriptor. */
200     pd->specific[0] = pd->specific_1stblock;
201
202     /* This is at least the second thread. */
203     pd->header.multiple_threads = 1;
204 #ifndef TLS_MULTIPLE_THREADS_IN_TCB
205     __pthread_multiple_threads = *__libc_multiple_threads_ptr = 1;
206 #endif
207
208 #ifdef NEED_DL_SYSINFO
209     /* Copy the sysinfo value from the parent. */
210     THREAD_SYSINFO(pd) = THREAD_SELF_SYSINFO;
211 #endif
212
213     /* The process ID is also the same as that of the caller. */
214     // 获取线程对应进程的pid
215     pd->pid = THREAD_GETMEM(THREAD_SELF, pid);
216
217     /* List of robust mutexes. */
218 #ifdef __PTHREAD_MUTEX_HAVE_PREV
219     pd->robust_list.__prev = &pd->robust_list;
220 #endif
221     pd->robust_list.__next = &pd->robust_list;
222
223     /* Allocate the DTV for this thread. */
224     if (_dl_allocate_tls(TLS_TPADJ(pd)) == NULL)
225     {
226         /* Something went wrong. */
227         assert(errno == ENOMEM);
228
229         /* Free the stack memory we just allocated. */
230         (void)munmap(mem, size);
231
232         return EAGAIN;
233     }
234 }
235
236

```

```

237     /* Prepare to modify global data. */
238     lll_lock(stack_cache_lock);
239
240     /* And add to the list of stacks in use. */
241     list_add(&pd->list, &stack_used);
242
243     lll_unlock(stack_cache_lock);
244
245     /* There might have been a race. Another thread might have
246        caused the stacks to get exec permission while this new
247        stack was prepared. Detect if this was possible and
248        change the permission if necessary. */
249     if (__builtin_expect((GL(dl_stack_flags) & PF_X) != 0 && (prot &
PROT_EXEC) == 0, 0))
250     {
251         int err = change_stack_perm(pd
252 #ifdef NEED_SEPARATE_REGISTER_STACK
253             ,
254             ~pagesize_m1
255 #endif
256         );
257         if (err != 0)
258         {
259             /* Free the stack memory we just allocated. */
260             (void)munmap(mem, size);
261
262             return err;
263         }
264     }
265
266     /* Note that all of the stack and the thread descriptor is
267        zeroed. This means we do not have to initialize fields
268        with initial value zero. This is specifically true for
269        the 'tid' field which is always set back to zero once the
270        stack is not used anymore and for the 'guardsize' field
271        which will be read next. */
272 }
273
274     /* Create or resize the guard area if necessary. */
275     if (__builtin_expect(guardsize > pd->guardsize, 0))
276     {
277 #ifdef NEED_SEPARATE_REGISTER_STACK
278         char *guard = mem + (((size - guardsize) / 2) & ~pagesize_m1);
279 #else
280         char *guard = mem;
281 #endif
282         if (mprotect(guard, guardsize, PROT_NONE) != 0)

```



```

283     {
284         int err;
285     mprot_error:
286         err = errno;
287
288         lll_lock(stack_cache_lock);
289
290         /* Remove the thread from the list. */
291         list_del(&pd->list);
292
293         lll_unlock(stack_cache_lock);
294
295         /* Get rid of the TLS block we allocated. */
296         _dl_deallocate_tls(TLS_TPADJ(pd), false);
297
298         /* Free the stack memory regardless of whether the size
299         of the cache is over the limit or not. If this piece
300         of memory caused problems we better do not use it
301         anymore. Uh, and we ignore possible errors. There
302         is nothing we could do. */
303         (void)munmap(mem, size);
304
305         return err;
306     }
307
308     pd->guardsize = guardsize;
309 }
310 else if (__builtin_expect(pd->guardsize - guardsize > size - reqsize,
311                          0))
312 {
313     /* The old guard area is too large. */
314
315     #ifdef NEED_SEPARATE_REGISTER_STACK
316         char *guard = mem + (((size - guardsize) / 2) & ~pagesize_m1);
317         char *oldguard = mem + (((size - pd->guardsize) / 2) & ~pagesize_m1);
318
319         if (oldguard < guard && mprotect(oldguard, guard - oldguard, prot) != 0)
320             goto mprot_error;
321
322         if (mprotect(guard + guardsize,
323                     oldguard + pd->guardsize - guard - guardsize,
324                     prot) != 0)
325             goto mprot_error;
326     #else
327         if (mprotect((char *)mem + guardsize, pd->guardsize - guardsize,
328                     prot) != 0)
329             goto mprot_error;

```

```

330 #endif
331
332     pd->guardsize = guardsize;
333 }
334 /* The pthread_getattr_np() calls need to get passed the size
335 requested in the attribute, regardless of how large the
336 actually used guardsize is. */
337     pd->reported_guardsize = guardsize;
338 }
339
340 /* Initialize the lock. We have to do this unconditionally since the
341 stillborn thread could be canceled while the lock is taken. */
342     pd->lock = LLL_LOCK_INITIALIZER;
343
344 /* We place the thread descriptor at the end of the stack. */
345 // 二级指针，返回struct thread的地址，其实就是一个堆快的地址，对比之前的示意图
346     *pdp = pd;
347
348 #if TLS_TCB_AT_TP
349 /* The stack begins before the TCB and the static TLS block. */
350     stacktop = ((char *) (pd + 1) - __static_tls_size);
351 #elif TLS_DTV_AT_TP
352     stacktop = (char *) (pd - 1);
353 #endif
354
355 #ifdef NEED_SEPARATE_REGISTER_STACK
356     *stack = pd->stackblock;
357     *stacksize = stacktop - *stack;
358 #else
359     *stack = stacktop;
360 #endif
361
362     return 0;
363 }
364
365 // 所以，在创建线程的时候，其实就是在pthread库内部，创建好描述线程的结构体对象，填充属性
366 // 第二步就是调用clone，让内核创建轻量级进程，并执行传入的回调函数和参数
367 // 其实，库提供的无非就是未来操作线程的API，通过属性设置线程的优先级之类，而真正调度的
368 // 过程，还是内核来的。
369 // 但是如果我们在上层，设计一些让线程暂停出让CPU，然后我们上次自定义队列，让线程的tcb
    进行排队
370 // 那么我们其实也可以基于内核，在用户层实现线程的调度，很多更高级的语言，可能会做这个工
    作。
371

```

## 6-2 线程栈

虽然 Linux 将线程和进程不加区分的统一到了 `task_struct`，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，简单理解就是main函数的栈空间，在fork的时候，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。如果扩充超出该上限则栈溢出会报段错误（发送段错误信号给该进程）。进程栈是唯一可以访问未映射页而不一定会发生段错误——超出扩充上限才报。
- 然而对于主线程生成的子线程而言，其 `stack` 将不再是向下生长的，而是事先固定下来的。线程栈一般是调用glibc/uclibc等的 `pthread` 库接口 `pthread_create` 创建的线程，在文件映射区（或称之为共享区）。其中使用 `mmap` 系统调用，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,  
2             MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
```

此调用中的 `size` 参数的获取很是复杂，你可以手工传入stack的大小，也可以使用默认的，一般而言就是默认的 `8M`。这些都不重要，重要的是，**这种stack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**。在glibc中通过mmap得到了stack之后，底层将调用 `sys_clone` 系统调用：

```
1 int sys_clone(struct pt_regs *regs)  
2 {  
3     unsigned long clone_flags;  
4     unsigned long newsp;  
5     int __user *parent_tidptr, *child_tidptr;  
6  
7     clone_flags = regs->bx;  
8     //获取了mmap得到的线程的stack指针  
9     newsp = regs->cx;  
10    parent_tidptr = (int __user *)regs->dx;  
11    child_tidptr = (int __user *)regs->di;  
12    if (!newsp)  
13        newsp = regs->sp;  
14    return do_fork(clone_flags, newsp, regs, 0, parent_tidptr, child_tidptr);  
15 }
```

因此，对于子线程的 `stack`，它其实是在进程的地址空间中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，是会浅拷贝生成者的 `task_struct` 的很多字段，如果愿意，其它线程也还是可以访问到的，于是一定要注意。