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

🍞 📌 不过:

- 仅仅有上面的理解，是不够的
- 要真正理解线程，就必须搞清楚，内核是如何进行资源划分的，尤其是代码

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：

物理内存

栈空间
数据段
代码段

程序 1

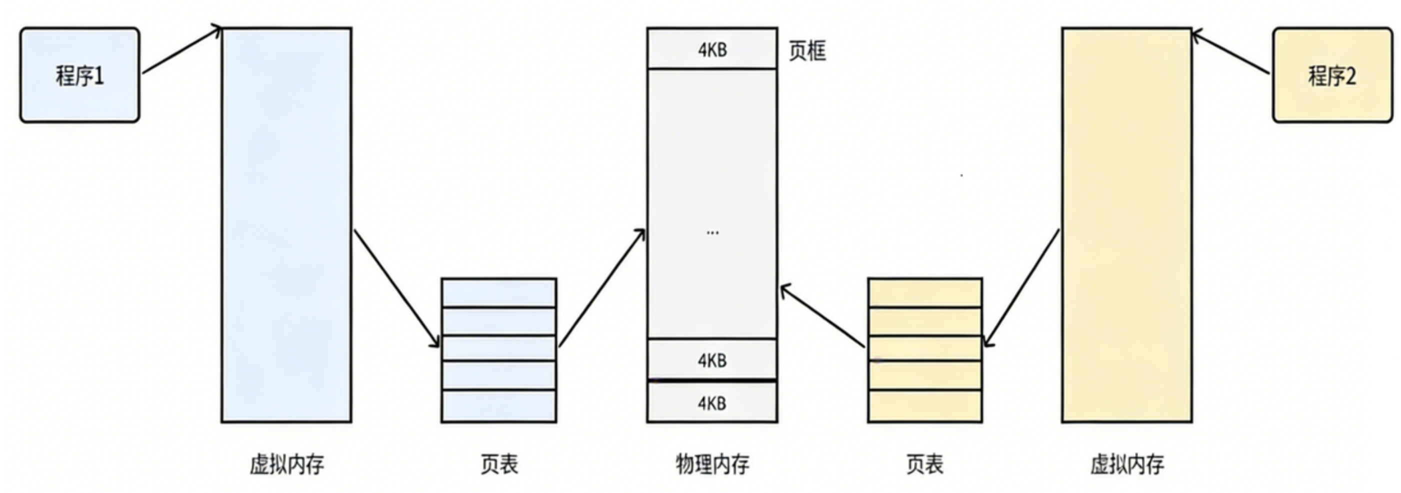
数据段
栈空间
栈空间
代码段
数据段
代码段

栈空间
数据段
代码段

程序 2

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 便并非是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。

代码块

```
1   /* include/linux/mm_types.h */
2   struct page
3    {
4       /* 原子标志，有些情况下会异步更新 */
5       unsigned long flags;
6       union
7       {
8           struct
9           {
10              /* 换出页列表，例如由zone->lru_lock保护的active_list */
11              struct list_head lru;
12              /* 如果最低为为0，则指向inode
13              * address_space，或为NULL
14              * 如果页映射为匿名内存，最低为置位
15              * 而且该指针指向anon_vma对象
16              */
17              struct address_space* mapping;
18              /* 在映射内的偏移量 */
19              pgoff_t index;
```

```
20              /*
21              * 由映射私有，不透明数据
22              * 如果设置了PagePrivate，通常用于buffer_heads
23              * 如果设置了PageSwapCache，则用于swp_entry_t
24              * 如果设置了PG_buddy，则用于表示伙伴系统中的阶
25              */
26              unsigned long private;
27          };
28
29          struct
30          {
31              /* slab, slob and slub */
32              union
33              {
34                  struct list_head slab_list; /* uses lru */
35                  struct
36                  {
37                      /* Partial pages */
38                      struct page* next;
39                      #ifdef CONFIG_64BIT
40                          int pages; /* Nr of pages left */
41                          int pobjects; /* Approximate count */
42                      #else
43                          short int pages;
44                          short int pobjects;
45                      #endif
46                  };
47              };
48
49              struct kmem_cache* slab_cache; /* not slob */
50              /* Double-word boundary */
51              void* freelist; /* first free object */
52              union
53              {
54                  void* s_mem; /* slab: first object */
55                  unsigned long counters; /* SLUB */
56                  struct
57                  { /* SLUB */
58                      unsigned inuse : 16; /* 用于SLUB分配器：对象的数目 */
59                      unsigned objects : 15;
60                      unsigned frozen : 1;
61                  };
62              };
63          };
64          ...
65      };
66
```

```
67        union
68        {
69            /* 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
70            搜索*/
71            atomic_t _mapcount;
72            unsigned int page_type;
73            unsigned int active; /* SLAB */
74            int units; /* SLOB */
75        };
76        ...
77    #if defined(WANT_PAGE_VIRTUAL)
78        /* 内核虚拟地址（如果没有映射则为NULL，即高端内存） */
79        void* virtual;
80    #endif /* WANT_PAGE_VIRTUAL */
81        ...
82    }
```

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。

```
#define PG_locked          0  /* Page is locked. Don't touch. */
#define PG_error           1
#define PG_referenced        2
#define PG_uptodate        3

#define PG_dirty           4
#define PG_lru             5
#define PG_active          6
#define PG_slab            7  /* slab debug (Suparna wants this) */

#define PG_checked         8  /* kill me in 2.5.<early>. */
#define PG_arch_1          9
#define PG_reserved       10
#define PG_private        11  /* Has something at ->private */

#define PG_writeback        12  /* Page is under writeback */
#define PG_nosave         13  /* Used for system suspend/resume */
#define PG_compound       14  /* Part of a compound page */
#define PG_swapcache        15  /* Swap page: swp_entry_t in private */

#define PG_mappedtodisk     16  /* Has blocks allocated on-disk */
#define PG_reclaim        17  /* To be reclaimed asap */
#define PG_nosave_free      18  /* Free, should not be written */
#define PG_buddy          19  /* Page is free, on buddy lists */
```

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，这个域的值为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。

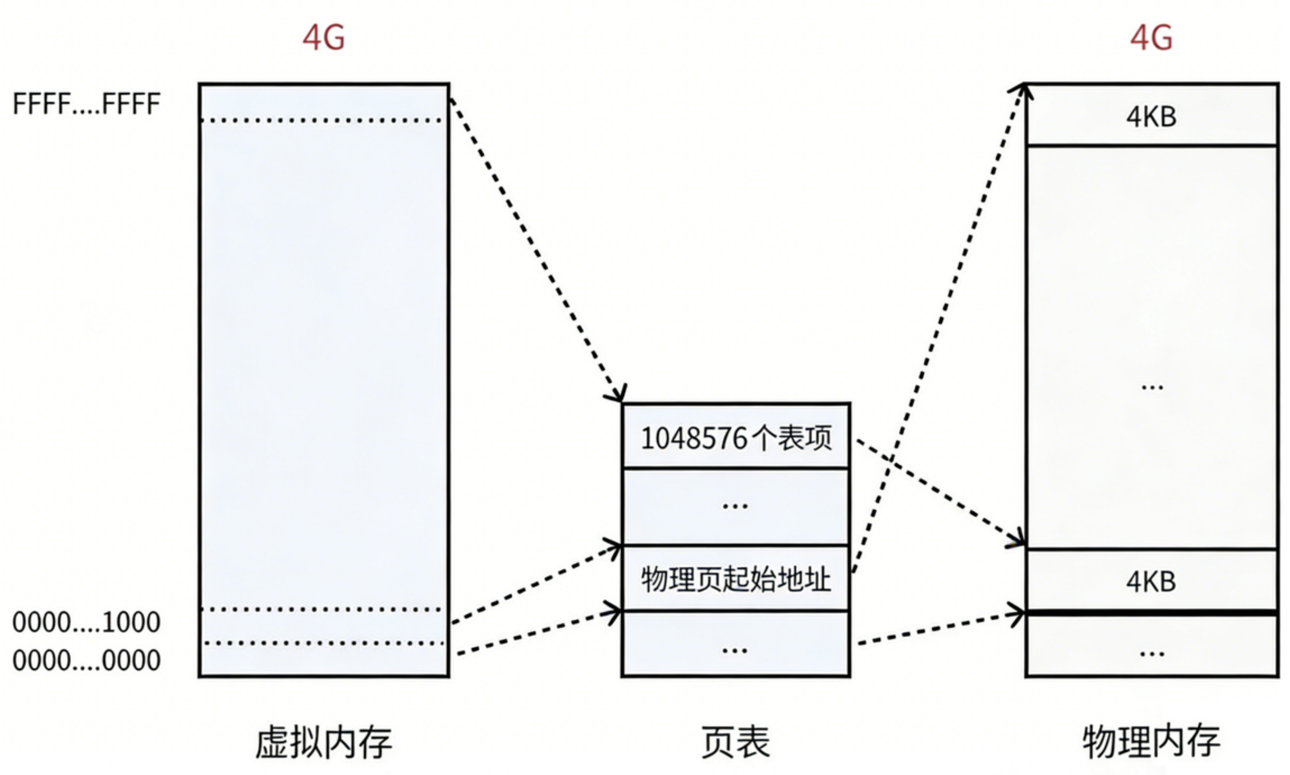
要知道的是，页的大小对于内存利用和系统开销来说非常重要，页太大，页内必然会剩余较大不能利用的空间（页内碎片）。页太小，虽然可以减小页内碎片的大小，但是页太多，会使得页表太长而占用内存，同时系统频繁地进行页转化，加重系统开销。因此，页的大小应该适中，通常为 512B - 8KB，windows/Linux系统的页框大小为4KB。

注意：

· 操作系统也要管理每一个页

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。

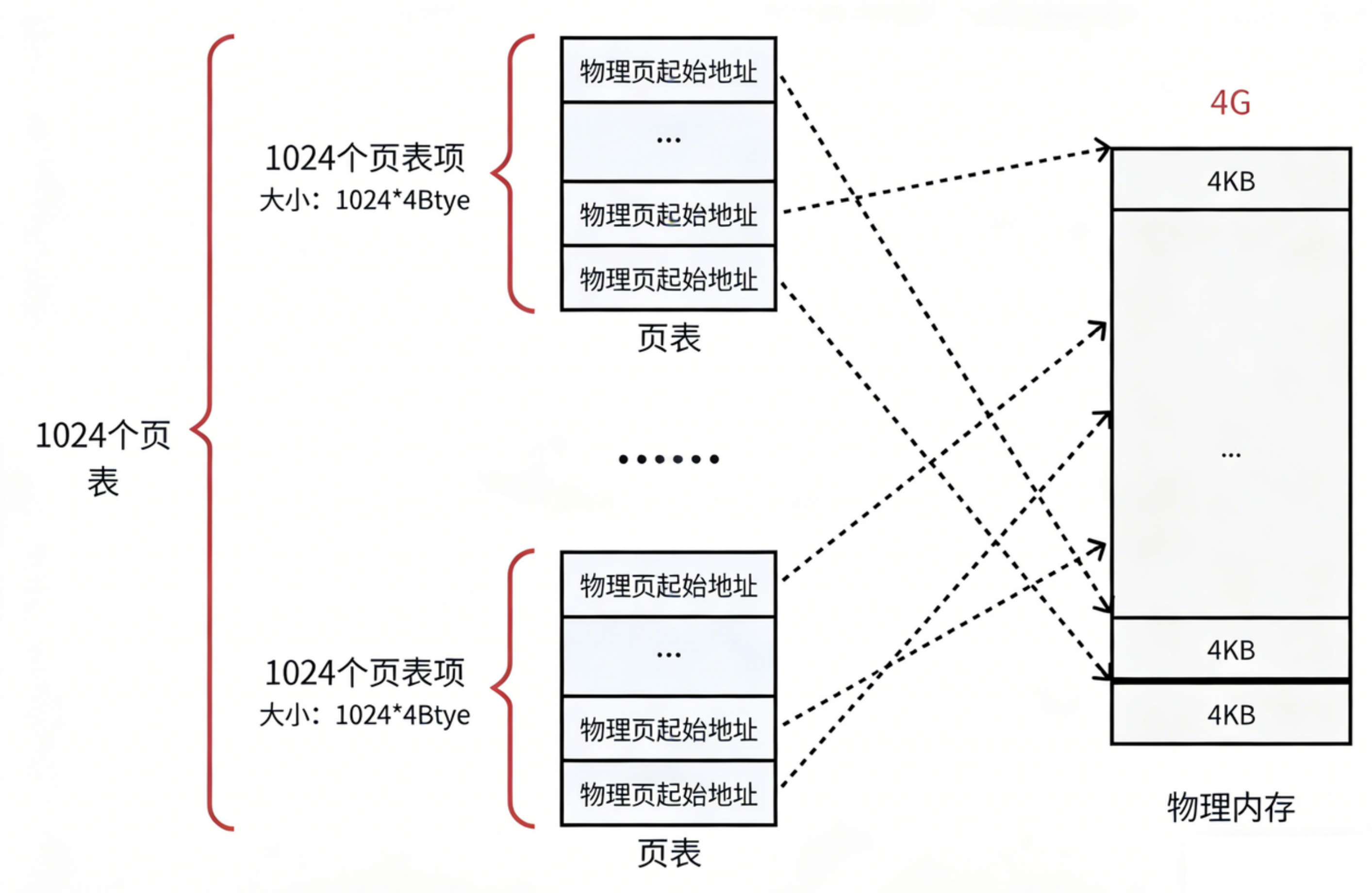
假设，在 32 位系统中，地址的长度是 4 个字节，那么页表中的每一个表项就是占用 4 个字节。所以

页表占据的总空间大小就是： `1048576*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(每个表中的表项个数) * 1024(表的个数)，仍然可以覆盖 4GB 的物理内存空间。



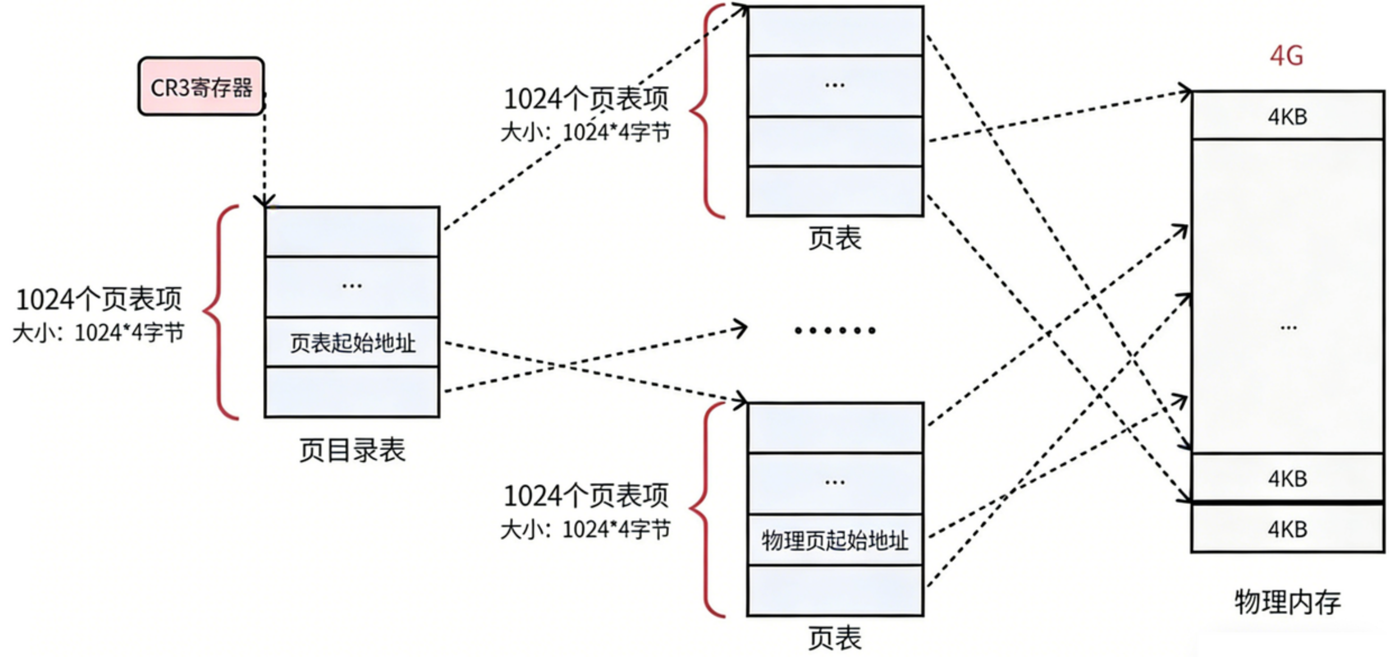
这里的每一个表，就是真正的页表，所以一共有 `1024` 个页表。一个页表自身占用 `4KB` ，那么 `1024` 个页表一共就占用了 `4MB` 的物理内存空间，和之前没差别啊？

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：



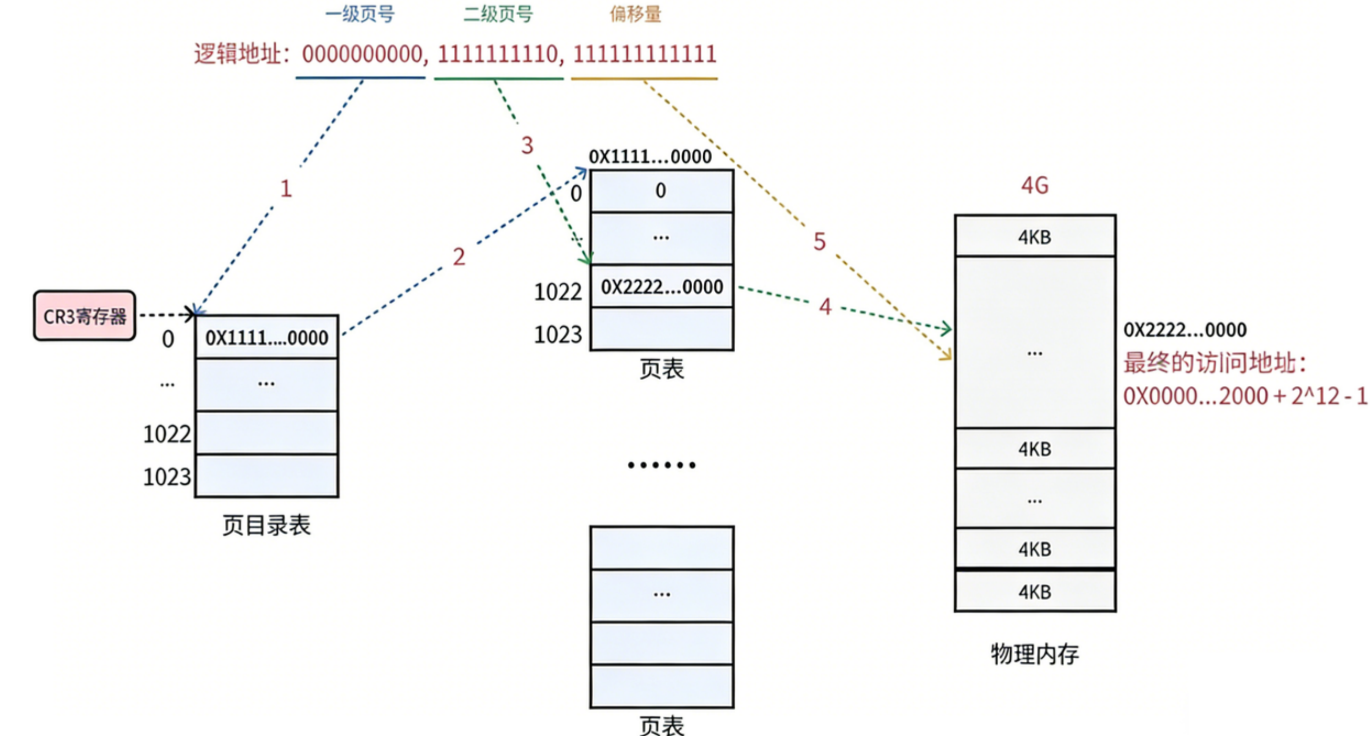
> · 所有页表的物理地址被页目录表项指向
>
> · 页目录的物理地址被 CR3 寄存器 指向，这个寄存器中，保存了当前正在执行任务的页目录地址。

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（0000000000,0000000001,11111111111）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。

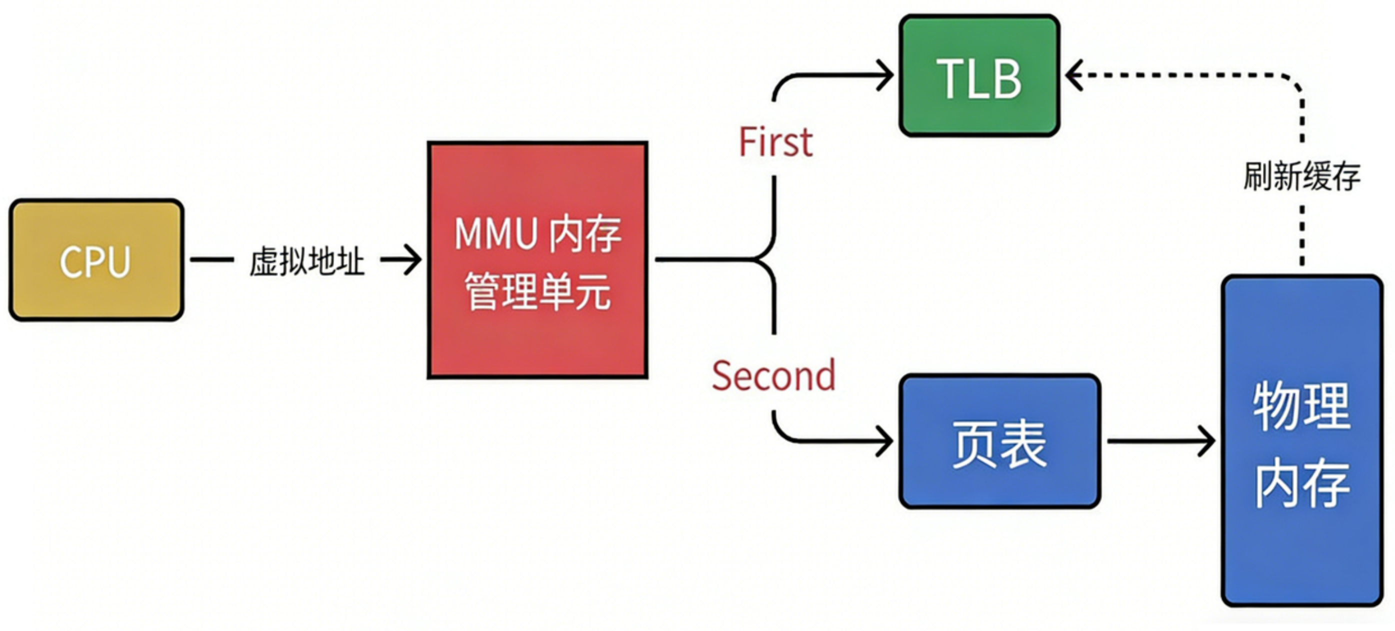


5. 注：一个物理页的地址一定是 4KB 对齐的(最后的 12 位全部为 0)，所以其实只需要记录物理页地址的高 20 位即可。
6. 以上其实就是 MMU 的工作流程。MMU(Memory Manage Unit)是一种硬件电路，其速度很快，主要工作是进行内存管理，地址转换只是它承接的业务之一。

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，Translation Lookaside Buffer，学名转译后备缓冲器）

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

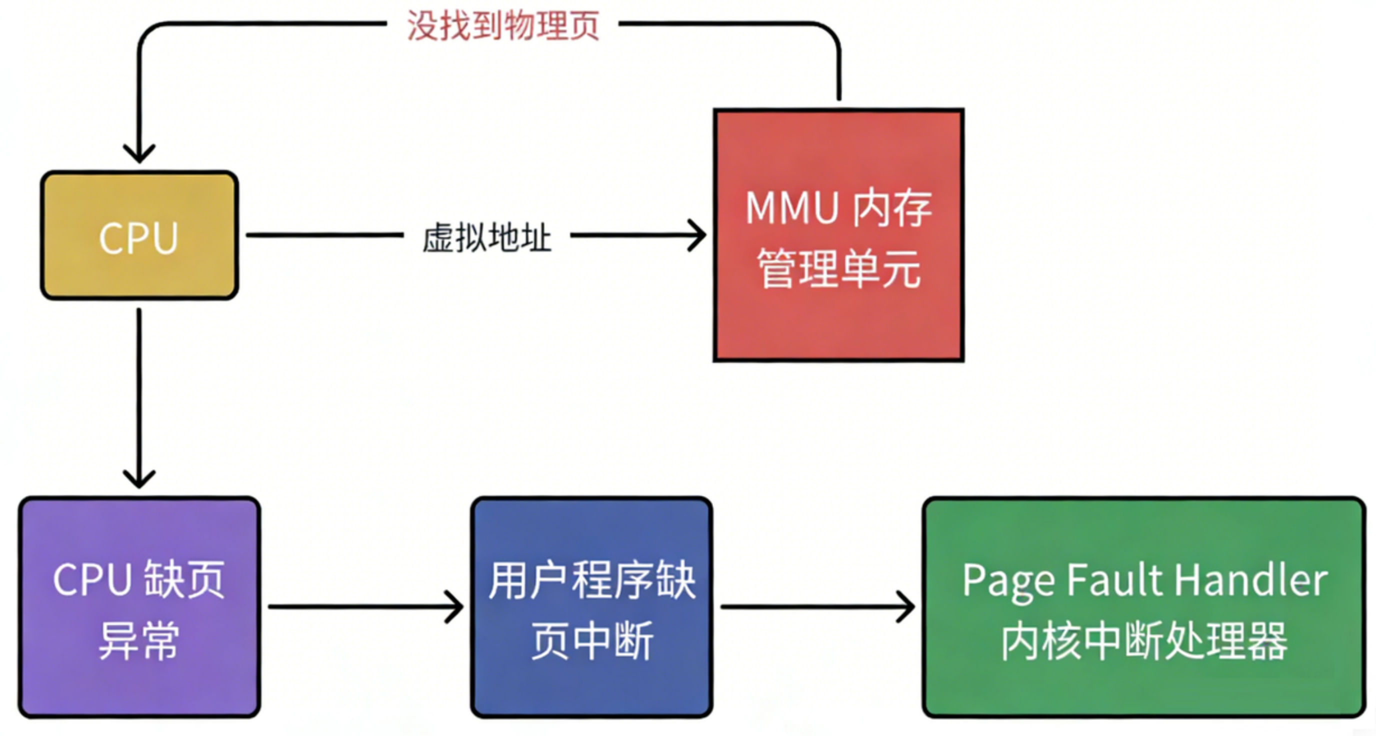
缺页异常 `Page Fault` ，它是一个由硬件中断触发的可以由软件逻辑纠正的错误。

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。

📌 注意：

- 如何理解我们之前的 new和malloc ?
- 如何理解我们之前学习的写时拷贝?
- 申请内存，究竟是在干什么?

如何区分是缺页了，还是真的越界了?

- 一个问题，越界了一定会报错吗?

1. 页号合法性检查：操作系统在处理中断或异常时，首先检查触发事件的虚拟地址的页号是否合法。如果页号合法但页面不在内存中，则为缺页中断；如果页号非法，则为越界访问。

2. 内存映射检查：操作系统还可以检查触发事件的虚拟地址是否在当前进程的内存映射范围内。如果地址在映射范围内但页面不在内存中，则为缺页中断；如果地址不在映射范围内，则为越界访问。

- 线程资源划分的真相：只要将虚拟地址空间进行划分，进程资源就天然被划分好了。

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
- 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 -- 哪些资源共享，哪些独占

- 进程间具有独立性
- 线程共享地址空间，也就共享进程资源

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"私有"数据:
>   - 线程ID
>   - 一组寄存器，线程的上下文数据
>   - 栈
>   - errno
>   - 信号屏蔽字
>   - 调度优先级

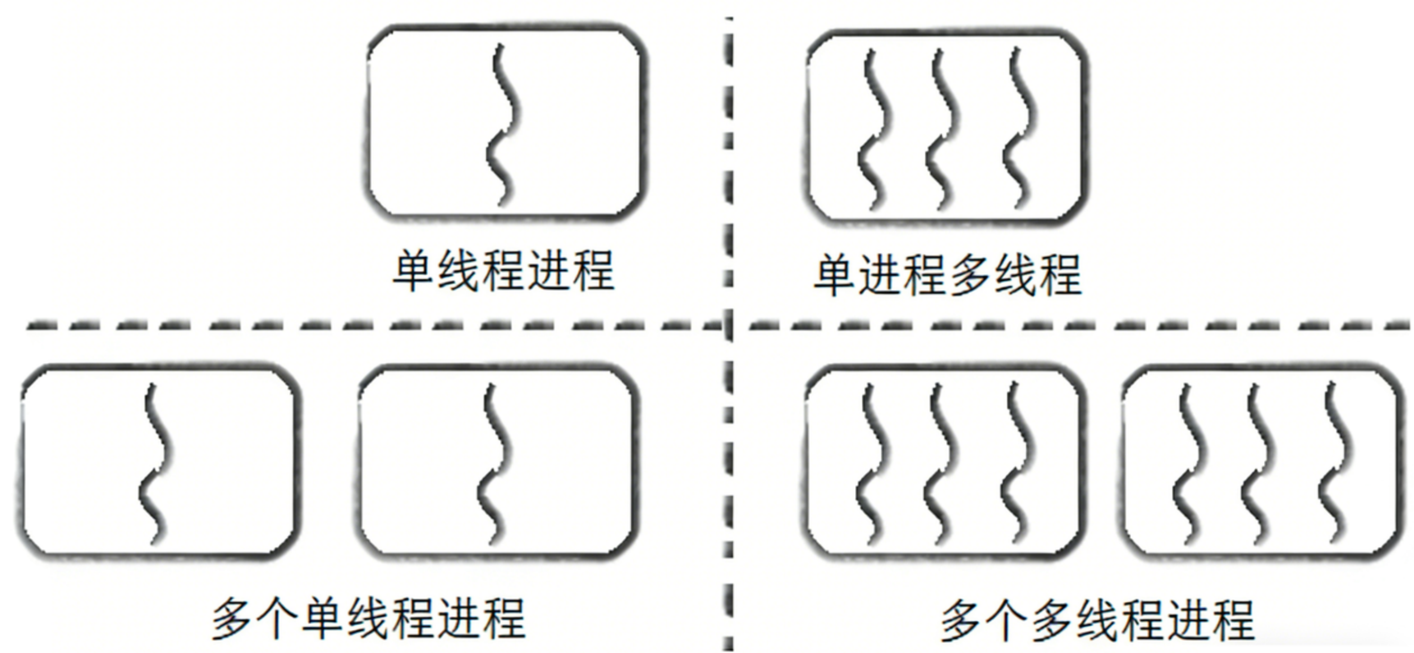
2.2 进程的多个线程共享

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

代码块

```
功能：创建一个新的线程
原型：
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
参数：
    thread:返回线程ID
    attr:设置线程的属性，attr为NULL表示使用默认属性
```

```
7       start_routine:是个函数地址，线程启动后要执行的函数
8       arg:传给线程启动函数的参数
9
10  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，建议通过返回值业判定，因为读取返回值要比读取线程内的errno变量的开销更小

代码块

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

```
31        }
32    }
```

代码块

```
1  #include <pthread.h>
2  // 获取线程ID
3  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：

代码块

```
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，而其他线程的栈在是在共享区（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

代码块

```
功能：线程终止

原型：
    void pthread_exit(void *value_ptr);

参数：
    value_ptr:value_ptr不要指向一个局部变量。

返回值：
    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

代码块

```
功能：取消一个执行中的线程

原型：
    int pthread_cancel(pthread_t thread);

参数：
    thread:线程ID

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。

代码块

```
功能：等待线程结束

原型
    int pthread_join(pthread_t thread, void **value_ptr);

参数:
    thread:线程ID
    value_ptr:它指向一个指针，后者指向线程的返回值

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:

代码块

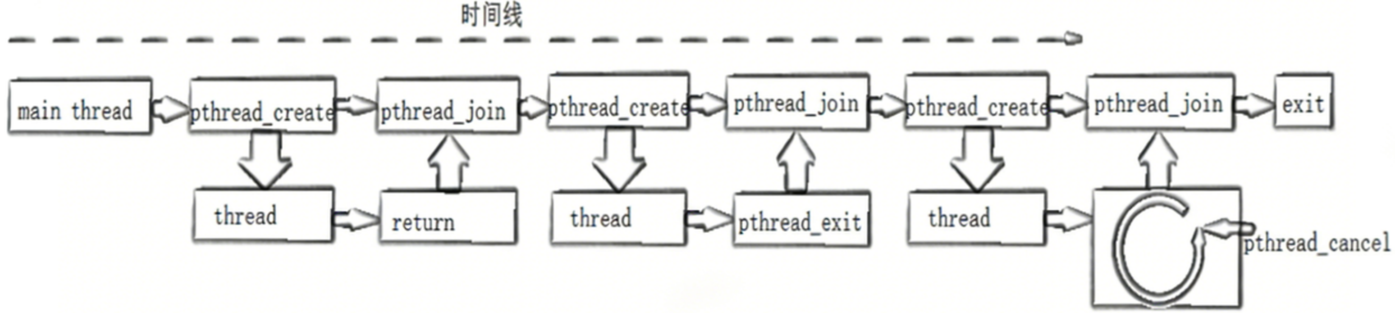
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
void *thread1( void *arg )
{
    printf("thread 1 returning ... \n");
    int *p = (int*)malloc(sizeof(int));
    *p = 1;

    return (void*)p;
}

void *thread2( void *arg )
{
    printf("thread 2 exiting ...\n");
    int *p = (int*)malloc(sizeof(int));
    *p = 2;
    pthread_exit((void*)p);
```

```
21  }
22  
23  void *thread3( void *arg )
24  {
25      while ( 1 )
26      { //
27          printf("thread 3 is running ...\n");
28          sleep(1);
29      }
30  
31      return NULL;
32  }
33  
34  int main( void )
35  {
36      pthread_t tid;
37      void *ret;
38      // thread 1 return
39      pthread_create(&tid, NULL, thread1, NULL);
40      pthread_join(tid, &ret);
41      printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
42      free(ret);
43  
44      // thread 2 exit
45      pthread_create(&tid, NULL, thread2, NULL);
46      pthread_join(tid, &ret);
47      printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
48      free(ret);
49  
50      // thread 3 cancel by other
51      pthread_create(&tid, NULL, thread3, NULL);
52      sleep(3);
53      pthread_cancel(tid);
54      pthread_join(tid, &ret);
55  
56      if ( ret == PTHREAD_CANCELED )
57          printf("thread return, thread id %X, return code:PTHREAD_CANCELED\n",
tid);
58      else
59          printf("thread return, thread id %X, return code:NULL\n", tid);
60  }
61  
62  运行结果:
63      [root@localhost linux]# ./a.out
64      thread 1 returning ...
65      thread return, thread id 5AA79700, return code:1
66      thread 2 exiting ...
```

```
67        thread return, thread id 5AA79700, return code:2
68        thread 3 is running ...
69        thread 3 is running ...
70        thread 3 is running ...
71        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。

代码块

```
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:

代码块

```
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。

代码块

```
1   #include <stdio.h>
2   #include <stdlib.h>
```

```c
3   #include <string.h>
4   #include <unistd.h>
5   #include <pthread.h>
6   void *thread_run( void * arg )
7   {
8       pthread_detach(pthread_self());
9       printf("%s\n", (char*)arg);
10      return NULL;
11  }
12
13  int main( void )
14  {
15      pthread_t tid;
16      if ( pthread_create(&tid, NULL, thread_run, "thread1 run...") != 0 )
17      {
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      if ( pthread_join(tid, NULL ) == 0 )
25      {
26          printf("pthread wait success\n");
27          ret = 0;
28      }
29
30      else
31      {
32          printf("pthread wait failed\n");
33          ret = 1;
34      }
35
36      return ret;
37  }
```

4. 线程ID及进程地址空间布局

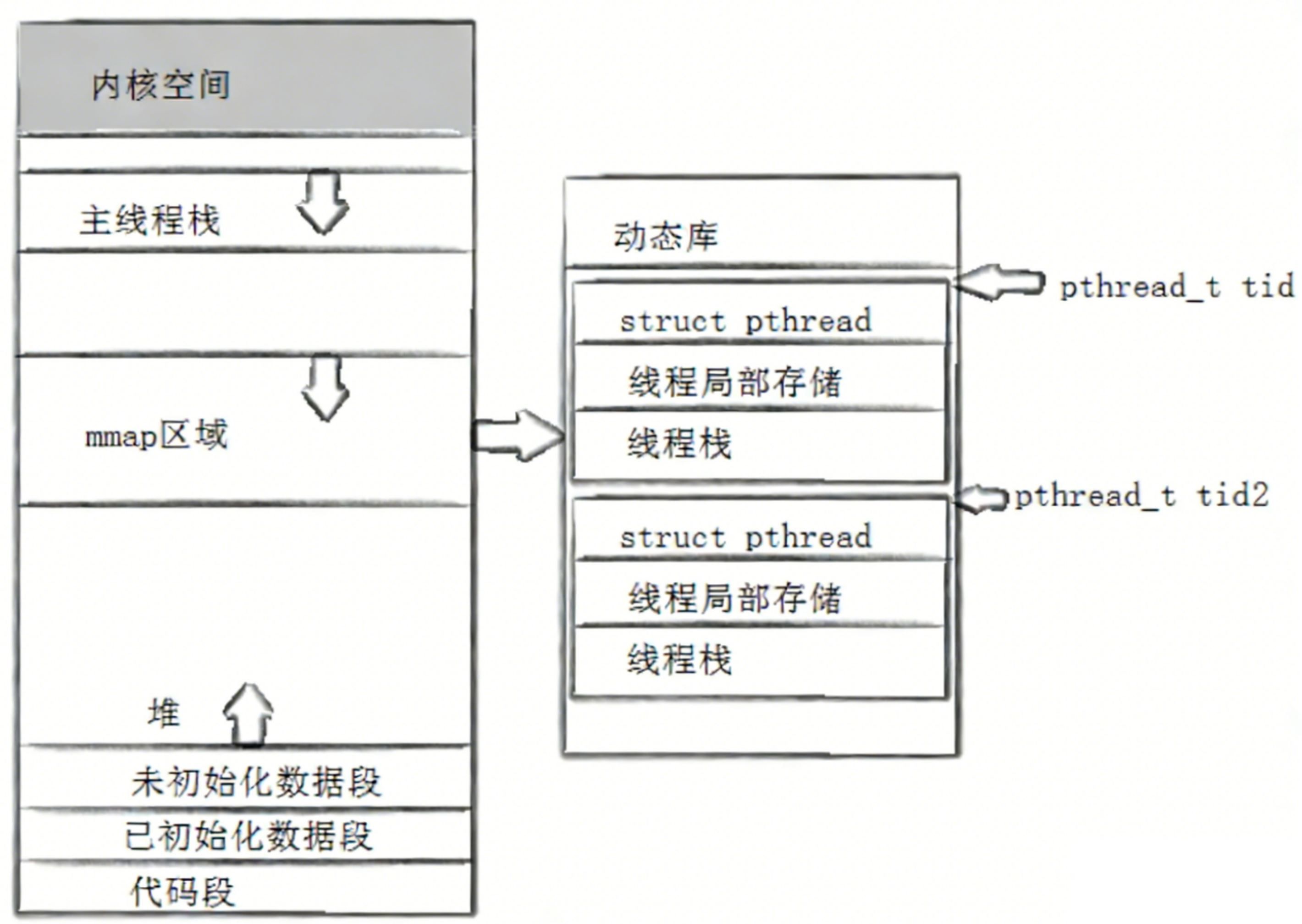
> · pthread_ create函数会产生一个线程ID，存放在第一个参数指向的地址中。该线程ID和前面说的线程ID不是一回事。
>
> · 前面讲的线程ID属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要一个数值来唯一表示该线程。
>
> · pthread_ create函数第一个参数指向一个虚拟内存单元，该内存单元的地址即为新创建线程的线程ID，属于NPTL线程库的范畴。线程库的后续操作，就是根据该线程ID来操作线程的。

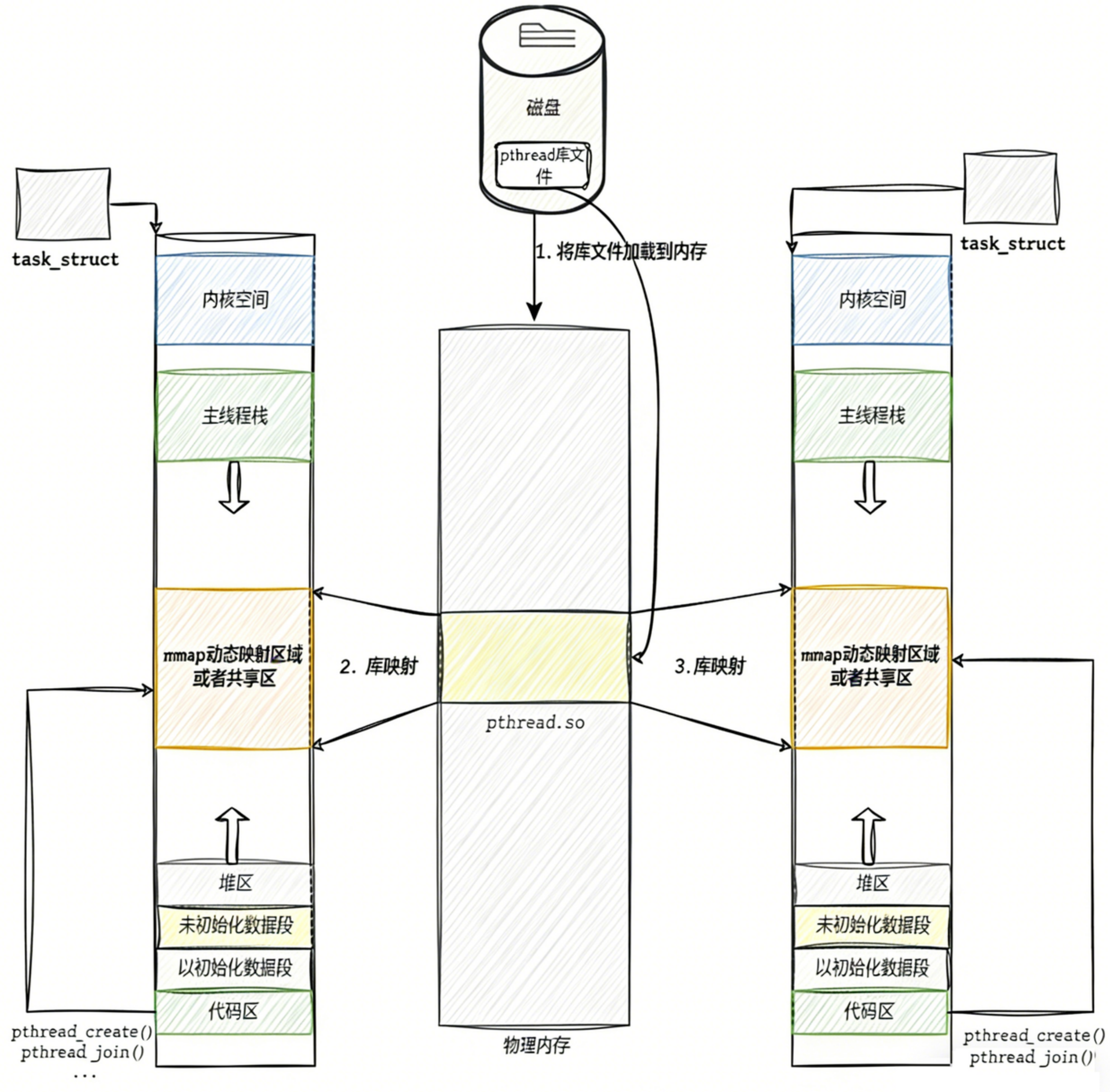
> · 线程库NPTL提供了pthread_ self函数，可以获得线程自身的ID：

代码块

```
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

代码块

```
1   // Thread.hpp
2   #pragma once
3   #include <iostream>
4   #include <string>
5   #include <functional>
6   #include <pthread.h>
7   namespace ThreadModule
8   {
9       // 原子计数器，方便形成线程名称
```

```
10      std::uint32_t cnt = 0;
11
12      // 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
13      using threadfunc_t = std::function<void()>;
14
15      // 线程状态
16      enum class TSTATUS
17      {
18      THREAD_NEW,
19      THREAD_RUNNING,
20      THREAD_STOP
21      };
22
23      // 线程
24      class Thread
25      {
26      private:
27          static void *run(void *obj)
28          {
29              Thread *self = static_cast<Thread *>(obj);
30              pthread_setname_np(pthread_self(), self->_name.c_str()); // 设置线程
    名称
31              self->_status = TSTATUS::THREAD_RUNNING;
32              if (!self->_joined)
33              {
34                  pthread_detach(pthread_self());
35              }
36              self->_func();
37              return nullptr;
38          }
39
40          void SetName()
41          {
42              // 后期加锁保护
43              _name = "Thread-" + std::to_string(cnt++);
44          }
45      public:
46          Thread(threadfunc_t func)
47              : _status(TSTATUS::THREAD_NEW)
48              , _joined(true), _func(func)
49          {
50              SetName();
51          }
52
53          void EnableDetach()
54          {
55              if (_status == TSTATUS::THREAD_NEW) _joined = false;
```

```
        }

        void EnableJoined()
        {
            if (_status == TSTATUS::THREAD_NEW) _joined = true;
        }

        bool Start()
        {
            if (_status == TSTATUS::THREAD_RUNNING) return true;
            int n = ::pthread_create(&_id, nullptr, run, this);
            if (n != 0) return false;
            return true;
        }

        bool Join()
        {
            if (_joined)
            {
                int n = pthread_join(_id, nullptr);
                if (n != 0) return false;
                return true;
            }
            return false;
        }
        ~Thread() {}

    private:
        std::string _name;
        pthread_t _id;
        TSTATUS _status;
        bool _joined;
        threadfunc_t _func;
    };
}
```

代码块

```
// main.cc
#include <iostream>
#include <unistd.h>
#include "test.hpp"
void hello1()
{
    char buffer[64];
    pthread_getname_np(pthread_self(), buffer, sizeof(buffer) - 1);
```

```
        while (true)
        {
            std::cout << "hello world, " << buffer << std::endl;
            sleep(1);
        }

        }
    void hello2()
    {
        char buffer[64];
        pthread_getname_np(pthread_self(), buffer, sizeof(buffer) - 1);
        while (true)
        {
            std::cout << "hello world, " << buffer << std::endl;
            sleep(1);
        }
    }

    int main()
    {
        pthread_setname_np(pthread_self(), "main");
        ThreadModule::Thread t1(hello1);
        t1.Start();

        ThreadModule::Thread t2(std::bind(&hello2));
        t2.Start();
        t1.Join();
        t2.Join();
        return 0;
    }
```

- pthread_setname_np 和 pthread_getname_np 是两个用于设置和获取线程名称的非标准函数（_np 表示 "non-portable"，即非可移植的）。它们通常在 Linux 和其他一些类 Unix 系统中可用，用于调试和多线程程序的管理
- 线程名称长度限制: 在 Linux 上，线程名称的最大长度为 16 个字符（包括结尾的 \0 ）。如果名称超过这个长度，会被截断。
- 权限: 通常，只有线程自身可以设置自己的名称。尝试设置其他线程的名称可能会导致错误。

代码块

```
// 运行结果查询
$ ps -aL
```

```
3       PID     LWP   TTY      TIME CMD
4   3195828 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
```

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

代码块

```
1   // 模版形式的
2   namespace ThreadModule
3   {
4       static int number = 1;
5       enum class TSTATUS
6       {
7           NEW,
8           RUNNING,
9           STOP
10      };
11
12      template <typename T>
13      class Thread
14      {
15      using func_t = std::function<void(T)>;
16      private:
17          // 成员方法!
18          static void *Routine(void *args)
19          {
20              Thread<T> *t = static_cast<Thread<T> *>(args);
21              t->_status = TSTATUS::RUNNING;
22              t->_func(t->_data);
23              return nullptr;
24          }
25
26          void EnableDetach() { _joinable = false; }
27
28      public:
29          Thread(func_t func, T data) : _func(func), _data(data),
30              _status(TSTATUS::NEW), _joinable(true)
31          {
32              _name = "Thread-" + std::to_string(number++);
```

```
33              _pid = getpid();
34          }
35
36          bool Start()
37          {
38              if (_status != TSTATUS::RUNNING)
39              {
40                  int n = ::pthread_create(&_tid, nullptr, Routine, this); //
41                  //TODO
42                  if (n != 0)
43                  return false;
44                  return true;
45              }
46              return false;
47          }
48
49          bool Stop()
50          {
51              if (_status == TSTATUS::RUNNING)
52              {
53                  int n = ::pthread_cancel(_tid);
54                  if (n != 0)
55                      return false;
56                  _status = TSTATUS::STOP;
57                  return true;
58              }
59              return false;
60          }
61
62          bool Join()
63          {
64              if (_joinable)
65              {
66                  int n = ::pthread_join(_tid, nullptr);
67                  if (n != 0) return false;
68                  _status = TSTATUS::STOP;
69                  return true;
70              }
71
72              return false;
73          }
74
75          void Detach()
76          {
77              EnableDetach();
78              pthread_detach(_tid);
79          }
```

```
80
81         bool IsJoinable() { return _joinable; }
82         std::string Name() { return _name; }
83
84         ~Thread()
85         {
86         }
87
88     private:
89         std::string _name;
90         pthread_t _tid;
91         pid_t _pid;
92         bool _joinable; // 是否是分离的，默认不是
93         func_t _func;
94         TSTATUS _status;
95         T _data;
96     };
97 }
```

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`

代码块

```
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 // 重点2: 传说中的原生线程库中的用来描述线程的tcb
15 struct pthread *pd = NULL;
16 // 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，
17 // struct pthread在空间的开头，一会追
```

```
18  int err = ALLOCATE_STACK(iattr, &pd);
19  if (__builtin_expect(err != 0, 0))
20  /* Something went wrong. Maybe a parameter of the attributes is
21  invalid or we could not allocate memory. */
22  versioned_symbol return err;
23  /* Initialize the TCB. All initializations with zero should be
24  performed in 'get_cached_stack'. This way we avoid doing this if
25  the stack freshly allocated with 'mmap'. */
26  #ifdef TLS_TCB_AT_TP
27  /* Reference to the TCB itself. */
28  pd->header.self = pd;
29  /* Self-reference for TLS. */
30  pd->header.tcb = pd;
31  #endif
32  /* Store the address of the start routine and the parameter. Since
33  we do not start the function directly the stillborn thread will
34  get the information from its thread descriptor. */
35  // 重点4: 向线程tcb中设置未来要执行的方法的地址和参数
36  pd->start_routine = start_routine;
37  pd->arg = arg;
38  /* Copy the thread attribute flags. */
39  struct pthread *self = THREAD_SELF;
40  pd->flags = ((iattr->flags & ~(ATTR_FLAG_SCHED_SET | ATTR_FLAG_POLICY_SET))
41  | (self->flags & (ATTR_FLAG_SCHED_SET | ATTR_FLAG_POLICY_SET)));
42  /* Initialize the field for the ID of the thread which is waiting
43  for us. This is a self-reference in case the thread is created
44  detached. */
45  pd->joinid = iattr->flags & ATTR_FLAG_DETACHSTATE ? pd : NULL;
46  /* The debug events are inherited from the parent. */
47  pd->eventbuf = self->eventbuf;
48  /* Copy the parent's scheduling parameters. The flags will say what
49  is valid and what is not. */
50  pd->schedpolicy = self->schedpolicy;
51  pd->schedparam = self->schedparam;
52  /* Copy the stack guard canary. */
53  #ifdef THREAD_COPY_STACK_GUARD
54  THREAD_COPY_STACK_GUARD(pd);
55  #endif
56  /* Copy the pointer guard value. */
57  #ifdef THREAD_COPY_POINTER_GUARD
58  THREAD_COPY_POINTER_GUARD(pd);
59  #endif
60  // 一堆参数设定，我们不关心
61  /* Determine scheduling parameters for the thread. */
62  if (attr != NULL && __builtin_expect((iattr->flags &
63  ATTR_FLAG_NOTINHERITSCHED) != 0, 0) && (iattr->flags & (ATTR_FLAG_SCHED_SET |
64  ATTR_FLAG_POLICY_SET)) != 0)
```

```
65  {
66  INTERNAL_SYSCALL_DECL(scerr);
67  /* Use the scheduling parameters the user provided. */
68  if (iattr->flags & ATTR_FLAG_POLICY_SET)
69  pd->schedpolicy = iattr->schedpolicy;
70  else if ((pd->flags & ATTR_FLAG_POLICY_SET) == 0)
71  {
72  pd->schedpolicy = INTERNAL_SYSCALL(sched_getscheduler, scerr, 1, 0);
73  pd->flags |= ATTR_FLAG_POLICY_SET;
74  }
75  if (iattr->flags & ATTR_FLAG_SCHED_SET)
76  memcpy(&pd->schedparam, &iattr->schedparam,
77  sizeof(struct sched_param));
78  else if ((pd->flags & ATTR_FLAG_SCHED_SET) == 0)
79  {
80  INTERNAL_SYSCALL(sched_getparam, scerr, 2, 0, &pd->schedparam);
81  pd->flags |= ATTR_FLAG_SCHED_SET;
82  }
83  /* Check for valid priorities. */
84  int minprio = INTERNAL_SYSCALL(sched_get_priority_min, scerr, 1,
85  iattr->schedpolicy);
86  int maxprio = INTERNAL_SYSCALL(sched_get_priority_max, scerr, 1,
87  iattr->schedpolicy);
88  if (pd->schedparam.sched_priority < minprio || pd-
89  >schedparam.sched_priority > maxprio)
90  {
91  err = EINVAL;
92  goto errout;
93  }
94  }
95  /* Pass the descriptor to the caller. */
96  // 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
97  *newthread = (pthread_t)pd;
98  /* Remember whether the thread is detached or not. In case of an
99  error we have to free the stacks of non-detached stillborn
100 threads. */
101 // 重点6：检测线程属性是否分离，这个很好理解
102 bool is_detached = IS_DETACHED(pd);
103 /* Start the thread. */
104 err = create_thread(pd, iattr, STACK_VARIABLES_ARGS); // 重点函数
105 if (err != 0)
106 {
107 /* Something went wrong. Free the resources. */
108 if (!is_detached)
109 {
110 errout:
111 __deallocate_stack(pd);
```

```
112 }
113 return err;
114 }
115 return 0;
116 }
117 // 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
118 __pthread_create_2_1
119 versioned_symbol(libpthread, __pthread_create_2_1, pthread_create, GLIBC_2_1);
```

线程属性：

代码块

```
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：

代码块

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

```
60  do { \
61  __pthread_slist_t *runp = THREAD_GETMEM (THREAD_SELF,
62  robust_list.__next);\
63  if (runp == &mutex->__data.__list) \
64  THREAD_SETMEM (THREAD_SELF, robust_list.__next, runp->__next); \
65  else \
66  { \
67  while (runp->__next != &mutex->__data.__list) \
68  runp = runp->__next; \
69  \
70  runp->__next = runp->__next->__next; \
71  mutex->__data.__list.__next = NULL; \
72  } \
73  } while (0)
74  #endif
75  /* List of cleanup buffers. */
76  struct _pthread_cleanup_buffer *cleanup;
77  /* Unwind information. */
78  struct pthread_unwind_buf *cleanup_jmp_buf;
79  #define HAVE_CLEANUP_JMP_BUF
80  /* Flags determining processing of cancellation. */
81  int cancelhandling;
82  /* Bit set if cancellation is disabled. */
83  #define CANCELSTATE_BIT 0
84  #define CANCELSTATE_BITMASK 0x01
85  /* Bit set if asynchronous cancellation mode is selected. */
86  #define CANCELTYPE_BIT 1
87  #define CANCELTYPE_BITMASK 0x02
88  /* Bit set if canceling has been initiated. */
89  #define CANCELING_BIT 2
90  #define CANCELING_BITMASK 0x04
91  /* Bit set if canceled. */
92  #define CANCELED_BIT 3
93  #define CANCELED_BITMASK 0x08
94  /* Bit set if thread is exiting. */
95  #define EXITING_BIT 4
96  #define EXITING_BITMASK 0x10
97  /* Bit set if thread terminated and TCB is freed. */
98  #define TERMINATED_BIT 5
99  #define TERMINATED_BITMASK 0x20
100 /* Bit set if thread is supposed to change XID. */
101 #define SETXID_BIT 6
102 #define SETXID_BITMASK 0x40
103 /* Mask for the rest. Helps the compiler to optimize. */
104 #define CANCEL_RESTMASK 0xffffff80
105 #define CANCEL_ENABLED_AND_CANCELED(value) \
106 (((value) & (CANCELSTATE_BITMASK | CANCELED_BITMASK | EXITING_BITMASK
```

```
107 \
108 | CANCEL_RESTMASK | TERMINATED_BITMASK)) == CANCELED_BITMASK)
109 #define CANCEL_ENABLED_AND_CANCELED_AND_ASYNCHRONOUS(value) \
110 (((value) & (CANCELSTATE_BITMASK | CANCELTYPE_BITMASK | CANCELED_BITMASK
111 \
112 | EXITING_BITMASK | CANCEL_RESTMASK | TERMINATED_BITMASK)) \
113 == (CANCELTYPE_BITMASK | CANCELED_BITMASK))
114 /* We allocate one block of references here. This should be enough
115 to avoid allocating any memory dynamically for most applications. */
116 struct pthread_key_data
117 {
118 /* Sequence number. We use uintptr_t to not require padding on
119 32- and 64-bit machines. On 64-bit machines it helps to avoid
120 wrapping, too. */
121 uintptr_t seq;
122 /* Data pointer. */
123 void *data;
124 } specific_1stblock[PTHREAD_KEY_2NDLEVEL_SIZE];
125 /* Two-level array for the thread-specific data. */
126 struct pthread_key_data *specific[PTHREAD_KEY_1STLEVEL_SIZE];
127 /* Flag which is set when specific data is set. */
128 bool specific_used;
129 /* True if events must be reported. */
130 bool report_events;
131 /* True if the user provided the stack. */
132 bool user_stack;
133 /* True if thread must stop at startup time. */
134 bool stopped_start;
135 /* Lock to synchronize access to the descriptor. */
136 lll_lock_t lock;
137 /* Lock for synchronizing setxid calls. */
138 lll_lock_t setxid_futex;
139 #if HP_TIMING_AVAIL
140 /* Offset of the CPU clock at start thread start time. */
141 hp_timing_t cpuclock_offset;
142 #endif
143 /* If the thread waits to join another one the ID of the latter is
144 stored here.
145 In case a thread is detached this field contains a pointer of the
146 TCB if the thread itself. This is something which cannot happen
147 in normal operation. */
148 struct pthread *joinid;
149 /* Check whether a thread is detached. */
150 #define IS_DETACHED(pd) ((pd)->joinid == (pd))
151 /* Flags. Including those copied from the thread attribute. */
152 int flags;
153 /* The result of the thread function. */
```

```
154 // 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
155 // 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
156 // 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
157 void *result;
158 /* Scheduling parameters for the new thread. */
159 struct sched_param schedparam;
160 int schedpolicy;
161 /* Start position of the code to be executed and the argument passed
162 to the function. */
163 // 用户指定的方法和参数
164 void *(*start_routine) (void *);
165 void *arg;
166 /* Debug state. */
167 td_eventbuf_t eventbuf;
168 /* Next descriptor with a pending event. */
169 struct pthread *nextevent;
170 #ifdef HAVE_FORCED_UNWIND
171 /* Machine-specific unwind info. */
172 struct _Unwind_Exception exc;
173 #endif
174 /* If nonzero pointer to area allocated for the stack and its
175 size. */
176 // 线程自己的栈和大小
177 void *stackblock;
178 size_t stackblock_size;
179 /* Size of the included guard area. */
180 size_t guardsize;
181 /* This is what the user specified and what we will report. */
182 size_t reported_guardsize;
183 /* Resolver state. */
184 struct __res_state res;
185 /* This member must be last. */
186 char end_padding[];
187 #define PTHREAD_STRUCT_END_PADDING \
188 (sizeof (struct pthread) - offsetof (struct pthread, end_padding))
189 } __attribute ((aligned (TCB_ALIGNMENT)));
```

create_thread

代码块

```
1 tatic 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   /* We rely heavily on various flags the CLONE function understands:
9   CLONE_VM, CLONE_FS, CLONE_FILES
10  These flags select semantics with shared address space and
11  file descriptors according to what POSIX requires.
12  CLONE_SIGNAL
13  This flag selects the POSIX signal semantics.
14  CLONE_SETTLS
15  The sixth parameter to CLONE determines the TLS area for the
16  new thread.
17  CLONE_PARENT_SETTID
18  The kernels writes the thread ID of the newly created thread
19  into the location pointed to by the fifth parameters to CLONE.
20  Note that it would be semantically equivalent to use
21  CLONE_CHILD_SETTID but it is be more expensive in the kernel.
22  CLONE_CHILD_CLEARTID
23  The kernels clears the thread ID of a thread that has called
24  sys_exit() in the location pointed to by the seventh parameter
25  to CLONE.
26  CLONE_DETACHED
27  No signal is generated if the thread exists and it is
28  automatically reaped.
29  The termination signal is chosen to be zero which means no signal
30  is sent. */
31  int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGNAL |
32  CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
33  #if __ASSUME_NO_CLONE_DETACHED == 0
34  | CLONE_DETACHED
35  #endif
36  | 0);
37  if (__builtin_expect(THREAD_GETMEM(THREAD_SELF, report_events), 0))
38  {
39  /* The parent thread is supposed to report events. Check whether
40  the TD_CREATE event is needed, too. */
41  const int _idx = __td_eventword(TD_CREATE);
42  const uint32_t _mask = __td_eventmask(TD_CREATE);
43  if ((_mask & (__nptl_threads_events.event_bits[_idx] | pd-
44  >eventbuf.eventmask.event_bits[_idx])) != 0)
45  {
46  /* We always must have the thread start stopped. */
47  pd->stopped_start = true;
48  /* Create the thread. We always create the thread stopped
49  so that it does not get far before we tell the debugger. */
50  int res = do_clone(pd, attr, clone_flags, start_thread,
51  STACK_VARIABLES_ARGS, 1);
52  if (res == 0)
```

```
53    {
54    /* Now fill in the information about the new thread in
55    the newly created thread's data structure. We cannot let
56    the new thread do this since we don't know whether it was
57    already scheduled when we send the event. */
58    pd->eventbuf.eventnum = TD_CREATE;
59    pd->eventbuf.eventdata = pd;
60    /* Enqueue the descriptor. */
61    do
62    pd->nextevent = __nptl_last_event;
63    while (atomic_compare_and_exchange_bool_acq(&__nptl_last_event,
64    pd, pd->nextevent) != 0);
65    /* Now call the function which signals the event. */
66    __nptl_create_event();
67    /* And finally restart the new thread. */
68    lll_unlock(pd->lock);
69    }
70    return res;
71    }
72    }
73    #ifdef NEED_DL_SYSINFO
74    assert(THREAD_SELF_SYSINFO == THREAD_SYSINFO(pd));
75    #endif
76    /* Determine whether the newly created threads has to be started
77    stopped since we have to set the scheduling parameters or set the
78    affinity. */
79    bool stopped = false;
80    if (attr != NULL && (attr->cpuset != NULL || (attr->flags &
81    ATTR_FLAG_NOTINHERITSCHED) != 0))
82    stopped = true;
83    pd->stopped_start = stopped;
84    /* Actually create the thread. */
85    int res = do_clone(pd, attr, clone_flags, start_thread,
86    STACK_VARIABLES_ARGS, stopped);
87    if (res == 0 && stopped)
88    /* And finally restart the new thread. */
89    lll_unlock(pd->lock);
90    return res;
91    }
```

do_clone

代码块

```
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   if (stopped)
10  /* We Make sure the thread does not run far by forcing it to get a
11  lock. We lock it here too so that the new thread cannot continue
12  until we tell it to. */
13  lll_lock(pd->lock);
14  /* One more thread. We cannot have the thread do this itself, since it
15  might exist but not have been scheduled yet by the time we've returned
16  and need to check the value to behave correctly. We must do it before
17  creating the thread, in case it does get scheduled first and then
18  might mistakenly think it was the only thread. In the failure case,
19  we momentarily store a false value; this doesn't matter because there
20  is no kosher thing a signal handler interrupting us right here can do
21  that cares whether the thread count is correct. */
22  atomic_increment(&__nptl_nthreads);
23  // 执行特性体系结构下的clone函数
24  if (ARCH_CLONE(fct, STACK_VARIABLES_ARGS, clone_flags,
25  pd, &pd->tid, TLS_VALUE, &pd->tid) == -1)
26  {
27  atomic_decrement(&__nptl_nthreads); /* Oops, we lied for a second. */
28  /* Failed. If the thread is detached, remove the TCB here since
29  the caller cannot do this. The caller remembered the thread
30  as detached and cannot reverify that it is not since it must
31  not access the thread descriptor again. */
32  if (IS_DETACHED(pd))
33  __deallocate_stack(pd);
34  return errno;
35  }
36  /* Now we have the possibility to set scheduling parameters etc. */
37  // 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
38  if (__builtin_expect(stopped != 0, 0))
39  {
40  INTERNAL_SYSCALL_DECL(err);
41  int res = 0;
42  /* Set the affinity mask if necessary. */
43  if (attr->cpuset != NULL)
44  {
45  res = INTERNAL_SYSCALL(sched_setaffinity, err, 3, pd->tid,
46  sizeof(cpu_set_t), attr->cpuset);
47  if (__builtin_expect(INTERNAL_SYSCALL_ERROR_P(res, err), 0))
48  {
49  /* The operation failed. We have to kill the thread. First
```

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

代码块

```
1 #define ARCH_CLONE __clone
2 __clone是glibc用汇编封装的一个调用clone系统调用的函数，所以
3 __clone的实现就是汇编，贴一份代码（sysdeps/unix/sysv/linux/x86_64）：
4 ENTRY (BP_SYM (__clone))
5 /* Sanity check arguments. */
6 movq $-EINVAL,%rax
7 testq %rdi,%rdi /* no NULL function pointers */
8 jz SYSCALL_ERROR_LABEL
9 testq %rsi,%rsi /* no NULL stack pointers */
10 jz SYSCALL_ERROR_LABEL
11 /* Insert the argument onto the new stack. */
12 subq $16,%rsi
13 movq %rcx,8(%rsi)
```

```
14 /* Save the function pointer. It will be popped off in the
15 child in the ebx frobbing below. */
16 movq %rdi,0(%rsi)
17 /* Do the system call. */
18 movq %rdx, %rdi
19 movq %r8, %rdx
20 movq %r9, %r8
21 movq 8(%rsp), %r10
22 movl $SYS_ify(clone),%eax // 获取系统调用号
23 /* End FDE now, because in the child the unwind info will be
24 wrong. */
25 cfi_endproc;
26 syscall // 陷入内核(x86_32是int 80)，要求内核创建轻量级进程
27 testq %rax,%rax
28 jl SYSCALL_ERROR_LABEL
29 jz L(thread_start)
30 这部分代码了解即可。
```

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

代码块

```
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 static int
6 allocate_stack(const struct pthread_attr *attr, struct pthread **pdp,
7 ALLOCATE_STACK_PARMS)
8 {
9 struct pthread *pd;
10 size_t size;
11 size_t pagesize_m1 = __getpagesize() - 1;
12 void *stacktop;
13 assert(attr != NULL);
14 assert(powerof2(pagesize_m1 + 1));
15 assert(TCB_ALIGNMENT >= STACK_ALIGN);
16 /* Get the stack size from the attribute if it is set. Otherwise we
17 use the default we determined at start time. */
18 size = attr->stacksize ?: __default_stacksize; // 获取栈大小，用户没设置就默认
19 /* Get memory for the stack. */
20 // 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
21 if (__builtin_expect(attr->flags & ATTR_FLAG_STACKADDR, 0))
22 {
23 uintptr_t adj;
24 /* If the user also specified the size of the stack make sure it
```

```
25  is large enough. */
26  if (attr->stacksize != 0 && attr->stacksize < (__static_tls_size +
27  MINIMAL_REST_STACK))
28  return EINVAL;
29  /* Adjust stack size for alignment of the TLS block. */
30  #if TLS_TCB_AT_TP
31  adj = ((uintptr_t)attr->stackaddr - TLS_TCB_SIZE) & __static_tls_align_m1;
32  assert(size > adj + TLS_TCB_SIZE);
33  #elif TLS_DTV_AT_TP
34  adj = ((uintptr_t)attr->stackaddr - __static_tls_size) &
35  __static_tls_align_m1;
36  assert(size > adj);
37  #endif
38  /* The user provided some memory. Let's hope it matches the
39  size... We do not allocate guard pages if the user provided
40  the stack. It is the user's responsibility to do this if it
41  is wanted. */
42  #if TLS_TCB_AT_TP
43  pd = (struct pthread *)((uintptr_t)attr->stackaddr - TLS_TCB_SIZE - adj);
44  #elif TLS_DTV_AT_TP
45  pd = (struct pthread *)(((uintptr_t)attr->stackaddr - __static_tls_size -
46  adj) - TLS_PRE_TCB_SIZE);
47  #endif
48  /* The user provided stack memory needs to be cleared. */
49  memset(pd, '\0', sizeof(struct pthread));
50  /* The first TSD block is included in the TCB. */
51  pd->specific[0] = pd->specific_1stblock;
52  /* Remember the stack-related values. */
53  pd->stackblock = (char *)attr->stackaddr - size;
54  pd->stackblock_size = size;
55  /* This is a user-provided stack. It will not be queued in the
56  stack cache nor will the memory (except the TLS memory) be freed. */
57  pd->user_stack = true;
58  /* This is at least the second thread. */
59  pd->header.multiple_threads = 1;
60  #ifndef TLS_MULTIPLE_THREADS_IN_TCB
61  __pthread_multiple_threads = *__libc_multiple_threads_ptr = 1;
62  #endif
63  #ifdef NEED_DL_SYSINFO
64  /* Copy the sysinfo value from the parent. */
65  THREAD_SYSINFO(pd) = THREAD_SELF_SYSINFO;
66  #endif
67  /* The process ID is also the same as that of the caller. */
68  pd->pid = THREAD_GETMEM(THREAD_SELF, pid);
69  /* List of robust mutexes. */
70  #ifdef __PTHREAD_MUTEX_HAVE_PREV
71  pd->robust_list.__prev = &pd->robust_list;
```

```
72  #endif
73  pd->robust_list.__next = &pd->robust_list;
74  /* Allocate the DTV for this thread. */
75  if (_dl_allocate_tls(TLS_TPADJ(pd)) == NULL)
76  {
77  /* Something went wrong. */
78  assert(errno == ENOMEM);
79  return EAGAIN;
80  }
81  /* Prepare to modify global data. */
82  lll_lock(stack_cache_lock);
83  /* And add to the list of stacks in use. */
84  list_add(&pd->list, &__stack_user);
85  lll_unlock(stack_cache_lock);
86  }
87  else
88  {
89  // 下面的都是库内部自己做的，我们关心的
90  /* Allocate some anonymous memory. If possible use the cache. */
91  size_t guardsize;
92  size_t reqsize;
93  void *mem;
94  const int prot = (PROT_READ | PROT_WRITE | ((GL(dl_stack_flags) & PF_X) ?
95  PROT_EXEC : 0));
96  #if COLORING_INCREMENT != 0
97  /* Add one more page for stack coloring. Don't do it for stacks
98  with 16 times pagesize or larger. This might just cause
99  unnecessary misalignment. */
100 if (size <= 16 * pagesize_m1)
101 size += pagesize_m1 + 1;
102 #endif
103 /* Adjust the stack size for alignment. */
104 size &= ~__static_tls_align_m1; // 设置空间对齐
105 assert(size != 0);
106 /* Make sure the size of the stack is enough for the guard and
107 eventually the thread descriptor. */
108 guardsize = (attr->guardsize + pagesize_m1) & ~pagesize_m1;
109 if (__builtin_expect(size < ((guardsize + __static_tls_size +
110 MINIMAL_REST_STACK + pagesize_m1) & ~pagesize_m1),
111 0))
112 /* The stack is too small (or the guard too large). */
113 return EINVAL;
114 /* Try to get a stack from the cache. */
115 // 先尝试从pthread缓存中申请空间
116 reqsize = size;
117 pd = get_cached_stack(&size, &mem);
118 if (pd == NULL)
```

```
119 {
120 /* To avoid aliasing effects on a larger scale than pages we
121 adjust the allocated stack size if necessary. This way
122 allocations directly following each other will not have
123 aliasing problems. */
124 #if MULTI_PAGE_ALIASING != 0
125 if ((size % MULTI_PAGE_ALIASING) == 0)
126 size += pagesize_m1 + 1;
127 #endif
128 // 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
129 // 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
130 mem = mmap(NULL, size, prot,
131 MAP_PRIVATE | MAP_ANONYMOUS | ARCH_MAP_FLAGS, -1, 0);
132 if (__builtin_expect(mem == MAP_FAILED, 0))
133 {
134 #ifdef ARCH_RETRY_MMAP
135 mem = ARCH_RETRY_MMAP(size);
136 if (__builtin_expect(mem == MAP_FAILED, 0))
137 #endif
138 return errno;
139 }
140 /* SIZE is guaranteed to be greater than zero.
141 So we can never get a null pointer back from mmap. */
142 assert(mem != NULL);
143 #if COLORING_INCREMENT != 0
144 /* Atomically increment NCREATED. */
145 unsigned int ncreated = atomic_increment_val(&nptl_ncreated);
146 /* We chose the offset for coloring by incrementing it for
147 every new thread by a fixed amount. The offset used
148 module the page size. Even if coloring would be better
149 relative to higher alignment values it makes no sense to
150 do it since the mmap() interface does not allow us to
151 specify any alignment for the returned memory block. */
152 size_t coloring = (ncreated * COLORING_INCREMENT) & pagesize_m1;
153 /* Make sure the coloring offsets does not disturb the alignment
154 of the TCB and static TLS block. */
155 if (__builtin_expect((coloring & __static_tls_align_m1) != 0, 0))
156 coloring = (((coloring + __static_tls_align_m1) & ~
157 (__static_tls_align_m1)) & ~pagesize_m1);
158 #else
159 /* Unless specified we do not make any adjustments. */
160 #define coloring 0
161 #endif
162 /* Place the thread descriptor at the end of the stack. */
163 // 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
164 thread(tcb)的地址
165 #if TLS_TCB_AT_TP
```

```c
166 pd = (struct pthread *)((char *)mem + size - coloring) - 1;
167 #elif TLS_DTV_AT_TP
168 pd = (struct pthread *)((((uintptr_t)mem + size - coloring -
169 __static_tls_size) & ~__static_tls_align_m1) - TLS_PRE_TCB_SIZE);
170 #endif
171 /* Remember the stack-related values. */
172 // 记录下来整个空间的地址和大小
173 pd->stackblock = mem;
174 pd->stackblock_size = size;
175 /* We allocated the first block thread-specific data array.
176 This address will not change for the lifetime of this
177 descriptor. */
178 pd->specific[0] = pd->specific_1stblock;
179 /* This is at least the second thread. */
180 pd->header.multiple_threads = 1;
181 #ifndef TLS_MULTIPLE_THREADS_IN_TCB
182 __pthread_multiple_threads = *__libc_multiple_threads_ptr = 1;
183 #endif
184 #ifdef NEED_DL_SYSINFO
185 /* Copy the sysinfo value from the parent. */
186 THREAD_SYSINFO(pd) = THREAD_SELF_SYSINFO;
187 #endif
188 /* The process ID is also the same as that of the caller. */
189 // 获取线程对应进程的pid
190 pd->pid = THREAD_GETMEM(THREAD_SELF, pid);
191 /* List of robust mutexes. */
192 #ifdef __PTHREAD_MUTEX_HAVE_PREV
193 pd->robust_list.__prev = &pd->robust_list;
194 #endif
195 pd->robust_list.__next = &pd->robust_list;
196 /* Allocate the DTV for this thread. */
197 if (_dl_allocate_tls(TLS_TPADJ(pd)) == NULL)
198 {
199 /* Something went wrong. */
200 assert(errno == ENOMEM);
201 /* Free the stack memory we just allocated. */
202 (void)munmap(mem, size);
203 return EAGAIN;
204 }
205 /* Prepare to modify global data. */
206 lll_lock(stack_cache_lock);
207 /* And add to the list of stacks in use. */
208 list_add(&pd->list, &stack_used);
209 lll_unlock(stack_cache_lock);
210 /* There might have been a race. Another thread might have
211 caused the stacks to get exec permission while this new
212 stack was prepared. Detect if this was possible and
```

```
213 change the permission if necessary. */
214 if (__builtin_expect((GL(dl_stack_flags) & PF_X) != 0 && (prot &
215 PROT_EXEC) == 0, 0))
216 {
217 int err = change_stack_perm(pd
218 #ifdef NEED_SEPARATE_REGISTER_STACK
219 ,
220 ~pagesize_m1
221 #endif
222 );
223 if (err != 0)
224 {
225 /* Free the stack memory we just allocated. */
226 (void)munmap(mem, size);
227 return err;
228 }
229 }
230 /* Note that all of the stack and the thread descriptor is
231 zeroed. This means we do not have to initialize fields
232 with initial value zero. This is specifically true for
233 the 'tid' field which is always set back to zero once the
234 stack is not used anymore and for the 'guardsize' field
235 which will be read next. */
236 }
237 /* Create or resize the guard area if necessary. */
238 if (__builtin_expect(guardsize > pd->guardsize, 0))
239 {
240 #ifdef NEED_SEPARATE_REGISTER_STACK
241 char *guard = mem + (((size - guardsize) / 2) & ~pagesize_m1);
242 #else
243 char *guard = mem;
244 #endif
245 if (mprotect(guard, guardsize, PROT_NONE) != 0)
246 {
247 int err;
248 mprot_error:
249 err = errno;
250 lll_lock(stack_cache_lock);
251 /* Remove the thread from the list. */
252 list_del(&pd->list);
253 lll_unlock(stack_cache_lock);
254 /* Get rid of the TLS block we allocated. */
255 _dl_deallocate_tls(TLS_TPADJ(pd), false);
256 /* Free the stack memory regardless of whether the size
257 of the cache is over the limit or not. If this piece
258 of memory caused problems we better do not use it
259 anymore. Uh, and we ignore possible errors. There
```

```
260 is nothing we could do. */
261 (void)munmap(mem, size);
262 return err;
263 }
264 pd->guardsize = guardsize;
265 }
266 else if (__builtin_expect(pd->guardsize - guardsize > size - reqsize,
267 0))
268 {
269 /* The old guard area is too large. */
270 #ifdef NEED_SEPARATE_REGISTER_STACK
271 char *guard = mem + (((size - guardsize) / 2) & ~pagesize_m1);
272 char *oldguard = mem + (((size - pd->guardsize) / 2) & ~pagesize_m1);
273 if (oldguard < guard && mprotect(oldguard, guard - oldguard, prot) != 0)
274 goto mprot_error;
275 if (mprotect(guard + guardsize,
276 oldguard + pd->guardsize - guard - guardsize,
277 prot) != 0)
278 goto mprot_error;
279 #else
280 if (mprotect((char *)mem + guardsize, pd->guardsize - guardsize,
281 prot) != 0)
282 goto mprot_error;
283 #endif
284 pd->guardsize = guardsize;
285 }
286 /* The pthread_getattr_np() calls need to get passed the size
287 requested in the attribute, regardless of how large the
288 actually used guardsize is. */
289 pd->reported_guardsize = guardsize;
290 }
291 /* Initialize the lock. We have to do this unconditionally since the
292 stillborn thread could be canceled while the lock is taken. */
293 pd->lock = LLL_LOCK_INITIALIZER;
294 /* We place the thread descriptor at the end of the stack. */
295 // 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
296 *pdp = pd;
297 #if TLS_TCB_AT_TP
298 /* The stack begins before the TCB and the static TLS block. */
299 stacktop = ((char *)(pd + 1) - __static_tls_size);
300 #elif TLS_DTV_AT_TP
301 stacktop = (char *)(pd - 1);
302 #endif
303 #ifdef NEED_SEPARATE_REGISTER_STACK
304 *stack = pd->stackblock;
305 *stacksize = stacktop - *stack;
306 #else
```

```
307    *stack = stacktop;
308    #endif
309    return 0;
310    }
311    // 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
312    // 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
313    // 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
314    // 过程，还是内核来的。
315    // 但是如果我们自己在上层，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
316    tcb进行排队
317    // 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
318    作。
```

`/* How to pass the values to the 'create_thread' function. */`

`#d#define STACK_VARIABLES_ARGS stackaddr //`
`STACK_VARIABLES_ARGS`

其实就是stack地址

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：
>
> 代码块
>
> ```
> 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：
>
> 代码块
>
> ```
> int sys_clone(struct pt_regs *regs)
> {
>     unsigned long clone_flags;
>     unsigned long newsp;
>     int __user *parent_tidptr, *child_tidptr;
>
>     clone_flags = regs->bx;
>     //获取了mmap得到的线程的stack指针
>     newsp = regs->cx;
>     parent_tidptr = (int __user *)regs->dx;
>     child_tidptr = (int __user *)regs->di;
>     if (!newsp)
>     newsp = regs->sp;
>     return do_fork(clone_flags, newsp, regs, 0, parent_tidptr,
>     child_tidptr);
> }
> ```
>
> 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，一定要注意。

6.3 页表和页表项

代码块

```
 * We keep two sets of PTEs - the hardware and the linux version.
 * This allows greater flexibility in the way we map the Linux bits
 * onto the hardware tables, and allows us to have YOUNG and DIRTY
 * bits.
 *
 * The PTE table pointer refers to the hardware entries; the "Linux"
 * entries are stored 1024 bytes below.
 */
// 页表标志位
#define L_PTE_PRESENT (1 << 0)
#define L_PTE_FILE (1 << 1) /* only when !PRESENT */
#define L_PTE_YOUNG (1 << 1)
#define L_PTE_BUFFERABLE (1 << 2) /* matches PTE */
#define L_PTE_CACHEABLE (1 << 3) /* matches PTE */
#define L_PTE_USER (1 << 4)
#define L_PTE_WRITE (1 << 5)
```

```
17 #define L_PTE_EXEC (1 << 6)
18 #define L_PTE_DIRTY (1 << 7)
19 #define L_PTE_COHERENT (1 << 9) /* I/O coherent (xsc3) */
20 #define L_PTE_SHARED (1 << 10) /* shared between CPUs (v6) */
21 #define L_PTE_ASID (1 << 11) /* non-global (use ASID, v6) */
22 // 页表是?
23 typedef struct { unsigned long pte; } pte_t; // 页表项
24 typedef struct { unsigned long pgd; } pgd_t; // 页全局目录项
```

代码块

```
1 pgd_t *
2 pgd_alloc(struct mm_struct *mm)
3 {
4 pgd_t *ret, *init;
5 ret = (pgd_t *)__get_free_page(GFP_KERNEL | __GFP_ZERO);
6 init = pgd_offset(&init_mm, 0UL);
7 if (ret) {
8 #ifdef CONFIG_ALPHA_LARGE_VMALLOC
9 memcpy (ret + USER_PTRS_PER_PGD, init + USER_PTRS_PER_PGD,
10 (PTRS_PER_PGD - USER_PTRS_PER_PGD - 1)*sizeof(pgd_t));
11 #else
12 pgd_val(ret[PTRS_PER_PGD-2]) = pgd_val(init[PTRS_PER_PGD-2]);
13 #endif
14 /* The last PGD entry is the VPTB self-map. */
15 pgd_val(ret[PTRS_PER_PGD-1])
16 = pte_val(mk_pte(virt_to_page(ret), PAGE_KERNEL));
17 }
18 return ret;
19 }
20 pte_t *
21 pte_alloc_one_kernel(struct mm_struct *mm, unsigned long address)
22 {
23 pte_t *pte = (pte_t *)__get_free_page(GFP_KERNEL|__GFP_REPEAT|__GFP_ZERO);
24 return pte;
25 }
26 struct mm_struct {
27 struct vm_area_struct * mmap; /* list of VMAs */
28 struct rb_root mm_rb;
29 struct vm_area_struct * mmap_cache; /* last find_vma result */
30 unsigned long (*get_unmapped_area) (struct file *filp,
31 unsigned long addr, unsigned long len,
32 unsigned long pgoff, unsigned long flags);
33 void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
34 unsigned long mmap_base; /* base of mmap area */
35 unsigned long task_size; /* size of task vm space */
```

```
36 unsigned long cached_hole_size; /* if non-zero, the largest hole
37 below free_area_cache */
38 unsigned long free_area_cache; /* first hole of size
39 cached_hole_size or larger */
40 pgd_t * pgd; // 页目录起始地址
41 }
```

6.4 可以进行任意参数传递的线程封装demo

代码块

```
1 #include <iostream>
2 #include <functional> // 用于 std::function 和 std::bind
3 #include <memory> // 用于 std::shared_ptr 和 std::unique_ptr
4 #include <pthread.h> // POSIX 线程库
5 #include <unistd.h>
6 class Thread {
7 public:
8 // 构造函数
9 Thread() : thread_id_(0), running_(false) {}
10 // 析构函数
11 ~Thread() {
12 if (running_) {
13 pthread_detach(thread_id_); // 分离线程，避免资源泄漏
14 }
15 }
16 // 启动线程，接受任意可调用对象和参数
17 template <typename Callable, typename... Args>
18 bool start(Callable&& func, Args&&... args) {
19 if (running_) {
20 std::cerr << "Thread is already running!" << std::endl;
21 return false;
22 }
23 // 将可调用对象和参数打包为一个 std::function<void()> 对象
24 // 使用 std::bind 将函数和参数绑定在一起
25 // std::forward 用于完美转发参数，保持参数的左值/右值属性
26 auto task = std::make_shared<std::function<void()>>(
27 std::bind(std::forward<Callable>(func), std::forward<Args>
28 (args)...)
29 );
30 // 将任务传递给线程入口函数
31 // 使用 new 在堆上分配 std::shared_ptr，确保任务对象在线程执行期间有效
32 if (pthread_create(&thread_id_, nullptr, &Thread::threadEntry, new
33 std::shared_ptr<std::function<void()>>(task)) != 0) {
34 std::cerr << "Failed to create thread!" << std::endl;
35 return false;
```

```
36 }
37 running_ = true;
38 return true;
39 }
40 // 等待线程结束
41 void join() {
42 if (running_) {
43 pthread_join(thread_id_, nullptr);
44 running_ = false;
45 }
46 }
47 private:
48 pthread_t thread_id_; // 线程 ID
49 bool running_; // 线程是否在运行
50 // 线程入口函数
51 static void* threadEntry(void* arg) {
52 // 从参数中提取任务并执行
53 // 使用 std::unique_ptr 管理 std::shared_ptr 的指针，确保资源释放
54 std::unique_ptr<std::shared_ptr<std::function<void()>>> task_ptr(
55 static_cast<std::shared_ptr<std::function<void()>>*>(arg)
56 );
57 auto task = *task_ptr; // 解引用获取任务对象
58 (*task)(); // 执行任务
59 return nullptr;
60 }
61 };
62 // 示例：测试函数
63 void printMessage(const std::string& message, int value, int a, int b, int c)
64 {
65 std::cout << "Message: " << message << ", Value: " << value << std::endl;
66 std::cout << "a:" << a << std::endl;
67 std::cout << "b:" << b << std::endl;
68 std::cout << "c:" << c << std::endl;
69 sleep(10);
70 }
71 int main() {
72 Thread thread;
73 // 启动线程，传递任意函数和参数
74 // 这里传递了一个普通函数 printMessage 和两个参数 "Hello, World!" 和 42
75 thread.start(printMessage, "Hello, World!", 42, 1, 2, 3);
76 // 等待线程结束
77 thread.join();
78 return 0;
79 }
```

代码块

```
$ ./a.out
Message: Hello, World!, Value: 42
a:1
b:2
c:3

$ ps -aL
PID LWP TTY TIME CMD
923509 923509 pts/1 00:00:00 a.out
923509 923510 pts/1 00:00:00 a.out
```

6.5 我们自己调用一下clone

- 直接AI形成

代码块

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#define STACK_SIZE (1024 * 1024) // 1MB 的栈空间

// 子进程执行的函数
static int child_func(void *arg)
{
    printf("Child process: PID = %d\n", getpid());
    return 0;
}

int main()
{
    char *stack = (char*)malloc(STACK_SIZE); // 为子进程分配栈空间

    if (stack == NULL)
    {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // 使用 clone 创建子进程
    pid_t pid = clone(child_func, stack + STACK_SIZE, CLONE_VM | SIGCHLD,
    NULL);
    if (pid == -1)
```

```c
30        {
31            perror("clone");
32            free(stack);
33            exit(EXIT_FAILURE);
34        }
35
36        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
37
38        // 等待子进程结束
39        if (waitpid(pid, NULL, 0) == -1)
40        {
41            perror("waitpid");
42            free(stack);
43            exit(EXIT_FAILURE);
44        }
45
46        free(stack);
47        return 0;
48    }
```