Erlang 运行时源码分析之——线程进度机制

郑思遥*

2013 年 1 月 27 日

摘要

线程进度机制是 Erts 5.9 引入的一项性能改进，是 Erlang 运行时面向多核处理器的优化措施之一。线程进度机制可以跟踪所有受管的线程的进度，以此判断线程是否完成了某个进度点的执行。Erlang 运行时中的系统阻塞功能被新的基于这项机制的系统阻塞功能替代。基于这项机制，Erlang 运行时还实现了无锁的同步数据结构，未来 Erlang 运行时会越来越多地采用无锁数据结构。本文分析了这项机制的基本原理，详细分析了运行时中相关的数据结构和源代码，还分析了全新系统阻塞功能的实现。

1 概述

线程进度跟踪机制（thread progress）是 Erts 5.9 引入的一个重要的内部改进，如 release notes 中提到的：

> The ERTS internal system block functionality has been replaced by new functionality for blocking the system. The old system block functionality had contention issues and complexity issues. The new functionality piggy-backs on thread progress tracking functionality needed by newly introduced lock-free synchronization in the runtime system. When the functionality for blocking the system isn't used, there is more or less no overhead at all. This since the functionality for tracking thread progress is there and needed anyway.
>
> ERTS 采用了全新的系统阻塞功能。之前的系统阻塞功能可能会产生争用的问题，而且非常复杂。新的系统阻塞功能依赖于“线程进度跟踪机制”，这也是运行时中新引入的无锁同步依赖的机制。如果没有使用系统阻塞的功能，那么几乎不会有什么开销，因为线程进度跟踪的机制总是在那。

这是 Erlang 运行时提升在多核（甚至众核）平台上性能的众多改进之一。本文分析线程进度跟踪机制的原理及其代码，并分析使用了线程进度跟踪机制的系统阻塞相关的 api 代码。下一篇博文将分析新引入的“无锁队列（lock-free queue）”的原理和代码，即上述引用中提到的无锁同步机制。

本文基于 R15B02 版本进行分析。

*zhengsyao@gmail.com, http://weibo.com/zhengsyao/

2 线程进度跟踪机制

无锁算法可以通过线程进度值来判断是否所有相关的线程都已经完成了执行的某个特定的进度点。Erts 中的线程分为两类：受管（managed）线程和非受管（unmanaged）线程。线程进度机制只跟踪受管线程的进度值。在 Erts 中，目前只有三类线程是受管线程：

1. 调度器线程（`erts/emulator/beam/erl_process.c/sched_thread_func()` 函数表示的线程）
2. 完成辅助工作的线程（aux 线程，`erts/emulator/beam/erl_process.c/aux_thread()` 函数表示的线程）
3. 系统消息分发线程（`erts/emulator/beam/erl_trace.c/sys_msg_dispatcher_func()` 函数表示的线程）

调度器线程的数目是可配置的，其他两种线程都只有 1 个，因此受管线程的总数是调度器数目 +2。受管线程都是 “行为良好” 的，可以保证以一定的频度更新自己的进度值。

非受管线程目前包括 Erlang 虚拟机中的异步线程，也就是虚拟机 +A 参数配置的异步线程池（async thread pool）中的线程，这些线程由于不能保证一定频度的更新进度，所以是不受管的线程。

每一个线程都有一个私有的进度值，还有一个全局的进度值。全局的进度值表示所有的线程都已经达到的进度值。下面是进度值的规则：

- 在受管的线程集合中，有且只有一个线程是 “领导（leader）” 线程
- 每一个受管线程都在固定的位置更新自己的进度值
- 领导线程除了更新自己的进度值之外，还要更新全局进度值
- 所有受管线程的进度值和全局进度值初始化为 0
- 线程运行到更新点的时候更新自己的进度值，但是这个进度值不超过全局进度值 +1
- 领导线程更新完了自己的进度值之后，要检查所有受管线程的进度值是否都达到了全局进度值 +1，如果达到了，则更新全局进度值 +1
- 如果没有领导线程，那么第一个发现这个事实的受管线程要抢先争当领导。当然如果有多个受管线程同时争当领导，要通过原子操作保证只有一个领导产生
- 如果有线程睡眠，那么这个线程要设置特殊的进度值，因此在睡觉的线程不会影响领导更新全局进度
- 如果线程进度值达到最大值了，则绕回到 0。由于上面的几条进度值规则，就算有线程的进度值绕回了，也不会影响进度之间的比较。在 Erts 中，用无符号的 64 位整数表示进度值

从以上规则可以看出，全局进度值小于等于所有线程当前的进度值。如果有线程运行很快，那么快线程的进度值更新一次之后不会再更新，而是会等全局进度值更新了之后再更新，所以可以看出线程之间的进度值最大相差 1。

不同类型的受管线程在固定的位置更新线程进度。目前调度器线程在以下时间点会更新线程进度：

- 前一个 Erlang 进程调度出之后，下一个 Erlang 进程调度执行之前
- 要进入睡眠的时候
- 唤醒的时候

辅助线程在完成一次辅助工作之后就更新一次线程进度。系统消息分发线程在每发送一条 trace 消息的时候就更新一次线程进度。

以上是线程进度跟踪机制的基本原理，下面分析线程进度跟踪机制模块提供的 api 以及具体的代码分析。本文只是介绍原理和分析代码，而没有介绍 Erlang 运行时如何通过这个机制实现无锁的同步机制，敬请期待后续的博文:)

3 线程进度跟踪机制模块提供的 api 及实现

线程进度跟踪机制相关的 api 和代码分布在头文件`erts/emulator/beam/erl_thr_process.h` 和实现文件`erts/emulator/beam/erl_thr_process.c` 中。下面分类列出了这个模块给整个 Erlang 运行时调用的 api，了解了这些 api 调用的作用之后就可以理解实现的代码了。

初始化类

- `void erts_thr_progress_pre_init(void)`：在`erl_init.c/early_init()` 中调用，做早期初始化，创建线程进度数据相关的 tsd key，初始化全局进度值为 0。
- `void erts_thr_progress_init(int no_schedulers, int managed, int unmanaged)`：初始化线程进度跟踪机制使用的全局数据结构，详见 3.1 小节对数据结构的描述。
- `void erts_thr_progress_register_managed_thread(ErtsSchedulerData *esdp, ErtsThrPrgrCallbacks *callbacks, int pref_wakeup)`：受管线程在系统中注册，登记回调函数，创建线程私有的进度数据，初始化数据结构。
- `void erts_thr_progress_register_unmanaged_thread(ErtsThrPrgrCallbacks * callbacks)`：非受管的线程在系统中注册，登记回调函数，创建线程私有的进度数据，初始化数据结构。

更新状态类

- `int erts_thr_progress_update(ErtsSchedulerData *esdp)`：受管线程更新自己的进度值。返回结果表明当前线程是否是领导线程。如果是领导线程，还需要调用`erts_thr_progress_leader_update`。
- `int erts_thr_progress_leader_update(ErtsSchedulerData *esdp)`：领导线程更新自己的进度值和全局进度值。
- `void erts_thr_progress_active(ErtsSchedulerData *esdp, int on)`
  `void erts_thr_progress_prepare_wait(ErtsSchedulerData *esdp)`
  `void erts_thr_progress_finalize_wait(ErtsSchedulerData *esdp)`：如果受管线程需要等待某个事件发生而需要进入睡眠状态，那么在睡眠之前，需要调用`erts_thr_progress_active` 将线程状态设置为非活跃，然后调用`erts_thr_progress_prepare_wait` 通知运行时进行线程睡眠之前的准备活动（例如设置线程的进度值为等待状态)。线程被唤醒之后，要及时调用`erts_thr_progress_finalize_wait` 通知运行时进行线程唤醒之后的活动（例如恢复线程的进度值)。然后调用`erts_thr_progress_active` 将线程状态设置为活跃。
- `void erts_thr_progress_wakeup(ErtsSchedulerData *esdp, ErtsThrPrgrVal value)`：受管线程和非受管线程都可以通过这个调用请求运行时在全局进度达到（或超越，不能保证准确地在达到的时候）给定值`value` 的时候唤醒自己。调用之后线程就可以在线程事件上睡觉等待被运行时唤醒。运行时会在内部数据结构中注册线程的请求，每次更新全局进度的时候如果发现到达了线程请求的进度值，则唤醒相应的线程。

系统阻塞类

- `void erts_thr_progress_block(void)`：调用的受管线程将其他受管线程阻塞。调用这个函数之后，其他受管线程在执行到下一次进度更新点的时候会发现这个线程的阻塞请求，从而进入阻塞状态。因为有时间差的存在，所以调用的受管线程会等待其他受管线程都已经进入了阻塞状态。这个函数返回的时候可以保证其他受管线程都已经成功阻塞。调用的受管线程可以执行一些排他的操作。

- `void erts_thr_progress_unblock(void)`：受管线程在执行完排他的操作之后，调用这个函数解除系统的阻塞，将其他被阻塞的受管线程唤醒。这是 Erts 5.9 引入的新的阻塞系统，替换了之前复杂且易产生争用的阻塞系统。
- `int erts_thr_progress_is_blocking(void)`：判断当前系统是否正在阻塞。显然这个 api 没多大作用。被阻塞的线程没机会调用，阻塞别人的线程自己还不知道是不是在阻塞么。目前 Erlang 运行时中只有一些调试代码使用了这个 api。

其他状态判断类

- `int erts_thr_progress_is_managed_thread(void)`：判断当前线程是否是受管线程。
- `ErtsThrPrgrVal erts_thr_progress_later(ErtsSchedulerData *)`：返回一个未来的进度值，当前还没有受管线程达到这个进度值。实际上对于受管线程，返回的是受管线程当前进度值 +2，对于非受管线程，返回的是当前全局值 +2。根据进度值的规则，这样可以保证返回的一定是一个未来的进度值，尽管不一定是最小的。
- `ErtsThrPrgrVal erts_thr_progress_current(void)`：返回当前的全局进度值。
- `int erts_thr_progress_has_reached_this(ErtsThrPrgrVal this, ErtsThrPrgrVal val)`：判断进度值 `val` 是否已经超越了进度值 `this`。
- `int erts_thr_progress_equal(ErtsThrPrgrVal val1, ErtsThrPrgrVal val2)`：判断进度值 `val1` 是否等于进度值 `val2`。
- `int erts_thr_progress_cmp(ErtsThrPrgrVal val1, ErtsThrPrgrVal val2)`：比较两个进度值 `val1` 和 `val2` 的关系。如果相等，则返回 0，如果 `val2` 超过了 `val1`，则返回 1，反之返回 -1。这个比较 api 考虑了进度值超过了最大值绕回的情况。
- `int erts_thr_progress_has_reached(ErtsThrPrgrVal val)`：判断当前的全局进度值是否达到了 `val`，同样考虑了绕回的情况。

以上就是线程进度模块提供的全部 api。了解了这些 api 实现的功能之后，我们就可以读懂具体的实现代码，在后面的代码分析中，我们会了解到 Rickard Green 大神使用到的各种优化技巧。下面首先分析这个模块使用到的数据结构。

3.1 数据结构和初始化

3.1.1 运行时的公共管理数据结构

下面是运行时线程进度跟踪机制使用的管理数据结构，这些数据是全局数据，图 1 展示了这些数据结构之间的关系。

```
1  typedef struct {
2     erts_atomic32_t len;
3     int id[1];
4  } ErtsThrPrgrManagedWakeupData;
5
6  typedef struct {
7     erts_atomic32_t len;
8     int high_sz;
9     int low_sz;
10    erts_atomic32_t *high;
11    erts_atomic32_t *low;
12 } ErtsThrPrgrUnmanagedWakeupData;
13
14 typedef struct {
15    erts_atomic32_t lflgs;
16    erts_atomic32_t block_count;
17    erts_atomic_t blocker_event;
```

```
18     erts_atomic32_t pref_wakeup_used;
19     erts_atomic32_t managed_count;
20     erts_atomic32_t managed_id;
21     erts_atomic32_t unmanaged_id;
22  } ErtsThrPrgrMiscData;
23
24  typedef struct {
25     ERTS_THR_PRGR_ATOMIC current;
26  } ErtsThrPrgrElement;
27
28  typedef union {
29     ErtsThrPrgrElement data;
30     char align__[ERTS_ALC_CACHE_LINE_ALIGN_SIZE(sizeof(ErtsThrPrgrElement))];
31  } ErtsThrPrgrArray;
32
33  typedef struct {
34     void *arg;
35     void (*wakeup)(void *);
36     void (*prepare_wait)(void *);
37     void (*wait)(void *);
38     void (*finalize_wait)(void *);
39  } ErtsThrPrgrCallbacks;
40
41  typedef struct {
42     union {
43        ErtsThrPrgrMiscData data;
44        char align__[ERTS_ALC_CACHE_LINE_ALIGN_SIZE(
45              sizeof(ErtsThrPrgrMiscData))];
46     } misc;
47     ErtsThrPrgrArray *thr;
48     struct {
49        int no;
50        ErtsThrPrgrCallbacks *callbacks;
51        ErtsThrPrgrManagedWakeupData *data[ERTS_THR_PRGR_WAKEUP_DATA_SIZE];
52     } managed;
53     struct {
54        int no;
55        ErtsThrPrgrCallbacks *callbacks;
56        ErtsThrPrgrUnmanagedWakeupData *data[ERTS_THR_PRGR_WAKEUP_DATA_SIZE];
57     } unmanaged;
58  } ErtsThrPrgrInternalData;
59
60  static ErtsThrPrgrInternalData *intrnl;
61  ErtsThrPrgr erts_thr_prgr__;
62  erts_tsd_key_t erts_thr_prgr_data_key__;
```

图中的灰色 padding 部分是填充区域，用于将所在的数据结构填满一条缓存线。从图中可以看出，`intrnl` 是所有数据结构的根，指向 `ErtsThrPrgrInternalData` 结构体。在 `ErtsThrPrgrInternalData` 结构体中：

- `misc` 的类型为 `ErtsThrPrgrMiscData`，顾名思义，就是一些不好分类的管理数据，后面会具体分析每一个字段的意义。
- `thr` 是指向 `ErtsThrPrgrArray` 数组的指针。这个数组中的每一项实际上就是 `ErtsThrPrgrElement` 填充满一条缓存线的内容，实际有用的数据是 64 位的原子变量 `current`。这个数组的长度等于受管线程的数目，每一项表示一个受管线程当前的进度值。在代码中，每一项值只能被其表示的那个受管线程写入，而其他线程只能读取。
- `managed` 字段管理了和受管线程相关的数据。
- `unmanaged` 字段管理了非受管线程相关的数据。后面会具体分析这两个数据结构。

`thr` 数组是一项重要的优化，在使用写回策略（write-back policy）的处理器上，线程对自己进度的更新甚至不需要写入内存。例如，假设有 4 个受管线程，线程 1 是领导，而且每一个线程都运行在一个处理器核心上，每一个处理器核心都有自己的私有缓存。所有线程都更新一次之后在每一个处理器核

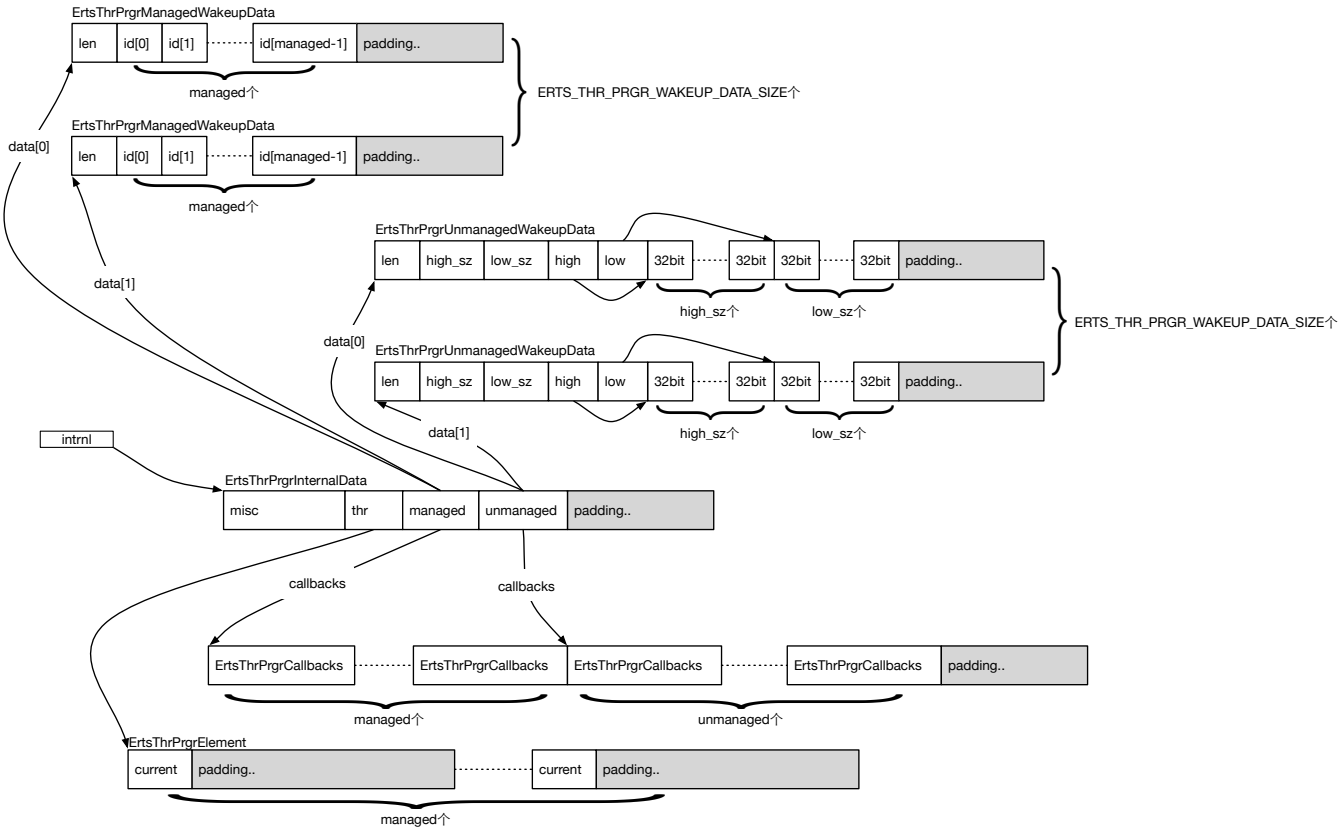

图 1: 线程进度跟踪机制使用的共享数据结构

心的私有缓存中都有一条完整的缓存线中保存的有用数据只包括该线程当前的进度值，将这种更新频繁的数据单独放在一条缓存线中可以避免伪共享。当领导线程要更新全局进度值的时候，需要读取所有受管线程的当前进度值。由于缓存一致性协议的作用，读取之后在领导线程所在的处理器核心的私有缓存中，会有每一个受管线程进度值所在的那条缓存线的副本。这时如果有一个普通受管线程要更新进度，如果缓存使用了写回策略（而不是写穿策略，write-through policy），那么这个线程所在的处理器核心会更新这个进度值所在的缓存线，并且缓存一致性协议通过缓存之间的高速网络通知领导所在的私有缓存这一个进度值所在的缓存线失效，除非这条缓存线被换出，否则不会将新的值写入内存。当领导需要访问所有受管线程的进度值的时候，发现对应的缓存线失效，缓存一致性协议会通过缓存之间的高速网络从原本的缓存线中直接获得最新的进度值。从此可以看出，在理想情况下，除了缓存预热之外，整个进度更新的过程都不需要读写内存，所有的数据访问都通过缓存一致性协议在处理器内部的高速网络上完成了，因此在核数很多的多核处理器上这个进度更新的机制也能高效率地工作。

`managed` 字段包含了两个数组：

- `ErtsThrPrgrCallbacks` 数组 `callbacks` 包含受管线程数目个元素，每一个元素是对应受管线程在注册的时候登记的回调函数，每一个线程都可以登记自己私有的回调函数。稍后会分析回调函数的作用。
- 指向 `ErtsThrPrgrManagedWakeupData` 指针的数组 `data`，一共有 `ERTS_THR_PRGR_WAKEUP_DATA_SIZE` 个元素，用于登记和请求唤醒相关的数据。稍后会详细分析这个数组的结构。

先说说回调函数 ErtsThrPrgrCallbacks 结构体。这个结构体就像一个闭包一样，包含 4 个回调函数和一个参数，这 4 个回调函数是：

- wakeup：用于唤醒该线程。
- prepare_wait：在睡该线程之前需要完成的准备工作。
- wait：将该线程置入睡眠状态。
- finalize_wait：唤醒该线程之后需要完成的恢复工作。

这些回调函数都和睡觉有关。结构体中包含的参数就是在调用这些回调函数的时候传递进去的参数。为什么需要这些回调函数呢？肯定有人会发现这些回调函数名字有些眼熟，那么这些回调函数和 erts_thr_progress_active、erts_thr_progress_prepare_wait、erts_thr_progress_finalize_wait 以及 erts_thr_progress_wakeup 这几个 api 函数的作用差别在哪呢？

既然是回调函数，那么从设计的角度来说，如果 B 模块向 A 模块提供回调函数 func，说明 func 是 A 需要让 B 执行的操作，但是 A 只知道需要进行这个操作，不知道这个操作的具体做法，所以具体的实现由 B 提供。那么在这里也是一样，回调函数是由系统阻塞功能使用的。当有线程调用 api 函数 erts_thr_progress_block 要求运行时阻塞系统的时候，运行时要让其他受管线程进入睡眠等待的状态，因而其他线程是被动进入睡眠状态。这里提供回调函数接口的目的是为了让线程被动睡眠前后能完成必要的数据维护操作。而 erts_thr_progress_active、erts_thr_progress_prepare_wait、erts_thr_progress_finalize_wait 以及 erts_thr_progress_wakeup 这几个 api 函数是线程自己因为种种原因需要主动进入睡眠状态的时候，通过调用这些 api 函数通知运行时，让运行时做好相关的数据维护操作。

这些回调函数调用时传入的参数一般都设置为对应线程的事件（thread-specific event，在代码中常缩写为 tse），因为 Erlang 运行时中通常通过事件等待机制实现线程的睡眠。

下面再看请求唤醒数据。请求唤醒是线程进度跟踪机制提供的一种功能，受管线程或非受管线程调用 erts_thr_progress_wakeup 请求运行时在特定的进度时唤醒线程。受管线程的请求数据就保存在 ErtsThrPrgrManagedWakeupData 数据结构中。从图 1 中可以看出，一个 ErtsThrPrgrManagedWakeupData 包含一个表示长度的 len 和一个 id 数组。len 表示后面这个 id 数组中有效元素的个数，从前往后每一个有效元素保存一个登记了唤醒信息受管线程的 id。data 数组的每一项表示一个特定进度值的唤醒信息。可是进度值的取值空间那么大（64 位无符号整数，也就是 2^{64}），那么这个唤醒信息的数组应该多大？这个数组是 ERTS_THR_PRGR_WAKEUP_DATA_SIZE 这么大，这个常量目前在 Erlang 虚拟机中定义为 4，只有 2^2。在线程请求唤醒的时候，运行时对给定的进度值进行掩码运算，只取了最后 2 个 bit，然后把线程 id 放进对应 ErtsThrPrgrManagedWakeupData 的 id 数组最后一个有效元素之后，并且增加 len 的值。这样，只要是指定进度值低位 2 个 bit 都相同的线程都会放在一起。每到一个新的全局进度值的时候，运行时会对当前全局进度值进行同样的掩码运算，得到一个索引，然后把这个索引对应一个 ErtsThrPrgrManagedWakeupData 中的有效 id 全部唤醒，通过调用这些线程的 wakeup 回调函数。很明显，每次唤醒的线程可能比实际需要唤醒的线程要多，但是线程唤醒之后可以重新检查睡眠等待的条件是否满足，如果不满足，继续睡眠。

unmanaged 字段也类似地包含了两个数组：回调函数数组 callbacks 和指向 ErtsThrPrgrUnmanagedWakeupData 指针的唤醒数据数组 data。

callbacks 数组的元素个数等于非受管线程的数目。由于非受管线程不参与系统阻塞功能，所以非受管线程在注册的时候一般不需要登记 prepare_wait、wait 和 finalize_wait 函数，而只需要 wakeup，因为非受管线程还可以使用请求唤醒机制，而这个机制调用 wakeup 回调函数唤醒线程。

非受管线程的唤醒数据 ErtsThrPrgrUnmanagedWakeupData 稍复杂一些，从图 1 中可以看出这个数据结构包含两个数组：low 和 high，这两个数组中每一个值都是 32 位原子值。low_sz 和 high_sz 分

别表示这两个数组的大小。len 还是表示对应的进度值有多少个请求唤醒的线程。

先看 low 数组的作用。在 Erlang 运行时中，异步线程就是非受管线程。Erlang 虚拟机允许通过 +A 参数指定异步线程的数目，目前这个参数允许取值范围为 0 到 1024，默认值为 0，未来在众核处理器上 +A 参数的上限可能还会调整，也就是说，非受管线程数目可能很大。所以这里通过 bitmap 来表示具体的线程。low 数组就是保存所有比特位的数组。一个线程对应一个比特位，low 数组中一个元素可以表示 32 个线程。所以这个数组一共需要 unmanaged/32 + 1 个元素。那么 high 数组保存的又是什么呢？high 数组中的 bit 对应 low 数组中的元素就好像 low 数组中的 bit 对应非受管线程的 id，也就是说，low 数组中的每一个元素在 high 数组中都有一个 bit 位对应。如果 low 数组中的某个元素中有 bit 被设置了，那么在 high 数组中也要设置相应的位，实现了二级索引。目前 Erlang 虚拟机最多允许 1024 个异步线程，1024/32 = 32，所以目前 high 数组中只用到了一个元素。

下面再提一下 misc 字段的内容。ErtsThrPrgrMiscData 结构体中包含以下字段：

- lflgs：标志位兼计数器。最高位是 ERTS_THR_PRGR_LFLG_BLOCK 标志，表示是否阻塞，接下来是 ERTS_THR_PRGR_LFLG_NO_LEADER 标志，表示是否还没有领导。剩下的 30 位是活跃线程的计数器。
- block_count：这个字段表示的意思并不是阻塞的计数器，而是系统阻塞功能中使用的一个计数器，表示没有阻塞的受管线程的数量。当这个计数器的值为 0 的时候，表示所有受管线程都阻塞了。
- blocker_event：阻塞者使用的事件。阻塞者在阻塞系统的时候，通过这个事件等待所有的受管线程都阻塞了。
- pref_wakeup_used：表示系统中是否已经设置了优先唤醒的受管线程。只能有一个线程是允许优先唤醒的，这个线程在线程进度跟踪机制中的 id 设置为 0。
- managed_count：受管线程的数目。
- managed_id 和unmanaged_id：在注册线程分配 id 的时候使用。

3.1.2 线程私有数据结构

下面介绍用于线程进度跟踪机制的线程私有数据结构。线程在注册的时候创建这个数据结构，并且以 erts_thr_prgr_data_key__ 作为键保存在自己的 TSD 中。这个数据简称 TPD（thread progress data）。

```
1  typedef struct {
2     int id;
3     int is_managed;
4     int is_blocking;
5     int is_temporary;
6     /* --- 以下字段是注册的线程专用的 --- */
7     ErtsThrPrgrVal wakeup_request[ERTS_THR_PRGR_WAKEUP_DATA_SIZE];
8     /* --- 以下字段是受管线程专用的 --- */
9     int leader;
10    int active;
11    struct {
12       ErtsThrPrgrVal local;
13       ErtsThrPrgrVal next;
14       ErtsThrPrgrVal current;
15    } previous;
16 } ErtsThrPrgrData;
```

这个数据结构各个字段的意义如下：

- id：这个线程在线程进度跟踪机制下的编号。
- is_managed：这个线程是否是受管线程。

- `is_blocking`：这个线程是否在阻塞系统[1]。
- `is_temporary`：表示这个 TPD 是否为临时数据。在系统 crash dump 的时候会用到临时 TPD，本文暂且不表。
- `wakeup_request`：保存这个线程发出的唤醒请求。
- `leader`：这个线程是否为领导线程。
- `active`：这个线程是否处于活跃状态。
- `previous`：这个字段用于计算进度值。`local` 和 `intrnl` 中 `thr` 数组中保存的 `current` 值保持一致。`next` 和 `current` 用于领导线程计算下一个全局进度值。

了解了原理和数据结构之后，读懂代码就不是什么太困难的事情了。作为多线程的程序，最困难的部分在于因为多个线程争用而互相产生干扰的情况的处理。下面几个小节主要分析一些代码中比较麻烦的部分。

4 线程进度跟踪机制代码分析

先看进度更新的代码。受管线程调用 `erts_thr_progress_update` 之后，这个函数调用 `update` 函数完成更新。如果受管线程是领导线程，则调用 `leader_update` 函数完成自己和全局的进度更新。在 `update` 函数中，如果通过检查`intrnl->misc.data.lflgs` 标志发现系统正在阻塞，则应该让 `leader_update` 阻塞线程。如果发现没有领导线程存在，那么要尝试通过一个原子操作去掉标志中的 `ERTS_THR_PRGR_LFLG_NO_LEADER`，如果操作成功，则说明自己成为了领导，而如果有并发线程也在抢夺领导的话肯定会失败。如果自己失败了，则不做任何操作，说明有并发线程正在抢夺领导，而且肯定会有线程抢夺成功。抢夺成功后，受管线程还应该调用 `leader_update` 函数进行全局更新。

在 `leader_update` 函数中，如果发现调用者并不是领导，说明要求阻塞，进入阻塞状态。`leader_update` 首先要像 `update` 那样对自己的进度进行更新。然后检查所有线程的当前进度值是否已经达到了下一个应该达到的进度值。如果都达到了，说明产生了进度，更新全局进度值，并且检查线程登记的请求唤醒信息，有的话则唤醒相应的线程。

5 系统阻塞机制的实现

系统阻塞通过是由 `thr_progress_block` 函数实现的。这个函数一开始有一个判断：

```
1 if (tpd->is_blocking++)
2    return (erts_aint32_t) 0;
```

把标志当计数器用。这是为了处理嵌套的情况。如果一个线程调用了一次阻塞之后，不知道自己已经阻塞了系统，然后又调用一次就可以直接返回。接下来一个 `while` 循环：

```
1 while (1) {
2    lflgs = erts_atomic32_read_bor_nob(&intrnl->misc.data.lflgs,
3                                       ERTS_THR_PRGR_LFLG_BLOCK);
4    if (lflgs & ERTS_THR_PRGR_LFLG_BLOCK)
5       block_thread(tpd);
6    else
7       break;
8 }
```

[1]注意语态，初看上去很容易猜测这个字段的意义是不是表示线程是否被阻塞。其实这里用的是主动语态。如果表示被阻塞，可能会采用 `is_blocked` 之类的名称。

这是为了防止有多个线程同时调用系统阻塞的情况。阻塞系统的那个线程负责设置 ERTS_THR_PRGR_LFLG_BLOCK 标志位。如果设置失败，则说明已经被别人抢先设置了，那么我自己只好先阻塞。唤醒之后，再去尝试设置标志位阻塞系统。如果设置成功了，则说明我成功获得了阻塞系统的权力，退出这个 while 循环继续后面的操作。

block_count_dec 函数和 block_count_inc 函数负责 intrnl->misc.data.block_count 计数器的递减和递增。block_count_dec 发现计数器归零的时候就知道大家都完成阻塞了，所以发送事件给阻塞者通知阻塞者唤醒。

6 请求唤醒机制的实现

线程请求唤醒的时候调用 erts_thr_progress_wakeup 函数，这个函数根据调用者是受管线程还是非受管线程分别调用 request_wakeup_managed 和 request_wakeup_unmanaged 函数。

request_wakeup_managed 函数有一个地方需要处理争用情况：

```
1 if (tpd->previous.local == value) {
2    value++;
3    if (value == ERTS_THR_PRGR_VAL_WAITING)
4       value = 0;
5
6    wix = ERTS_THR_PRGR_WAKEUP_IX(value);
7    if (tpd->wakeup_request[wix] == value)
8       return; /* Already got a request registered */
9 }
```

如果发现请求唤醒的进度值刚好等于线程当前的进度值，那么要擅自将请求唤醒的进度值向前步进 1。这是为了防止在注册进度值的时候全局进度已经达到这个值了，以免失去被唤醒的机会。将请求唤醒的进度值增加 1 是安全的，因为全局进度步进到下一个值的时候最多等于这个线程的当前值，而在写入请求的这段时间，这个线程不可能执行更新进度的代码，所以全局进度肯定不会更新到这个线程当前值的下一个值。所以写完请求值之后这个线程一定会被唤醒。

唤醒机制另一个麻烦一点的地方就是非受管线程的多级唤醒数据，但是了解了原理之后代码也很容易看懂了。