13 | 标准库：你需要了解的 C 并发编程基础知识有哪些？

2022-01-12 于航

讲述：于航
时长 16:55 大小 15.51M

你好，我是于航。

在构建高性能应用时，并发编程是我们经常采用的一种技巧。它通过在程序的运行进程内提供可控制粒度更细的“线程”，从而将程序的整体功能拆分为更小的独立任务单元，并以此来进一步利用多核 CPU 的运算资源。

对于 C11 标准之前的 C 语言来说，想要构建多线程应用，只能依赖于所在平台上的专有接口，比如 Unix 与类 Unix 平台上广泛使用的 POSIX 模型，以及后起之秀 OpenMP 模型等。这些模型提供的编程接口，以及所支持平台都有很大的不同。因此，对于那时的 C 语言来说，想要编写高可移植性的多线程应用，仍需要花费很大功夫。

而自 C11 标准后，C 语言为我们专门提供了一套通用的并发编程接口，你可以通过标准库头文件 threads.h 与 stdatomic.h 来使用它们。其中，threads.h 中包含有与线程控制、互斥量、条件变量，以及线程本地存储相关的接口；而 stdatomic.h 中则包含有与原子操作相关的接

口。这些接口提供了多种不同方式，可用来避免多线程应用在运行过程中可能遇到的各类线程同步问题。

C11 标准的发布，理论上使构建可移植的多线程 C 应用成为可能，但现实情况却并非这样理想。各类 C 标准库对 C11 中并发编程接口的支持程度不同，比如 Glibc（GNU C 标准库）在其 2018 年中旬发布的 2.28 版本中，才将相关接口进行了较为完整的实现。这就导致了 C11 标准中的这些重要特性，至今（2022 年初）仍然没有得到较为广泛的应用。

因此，接下来我将用两讲的篇幅，为你从零介绍有关并发编程的一些基础知识，以及 C11 并发编程相关接口的基本使用方式。当然，由于并发编程是一种完全不同的编程模型，短短几千字是无法详细梳理完所有相关内容的。但我希望通过对这两讲内容的学习，你能够对使用 C 语言开发多线程应用有一个全新的认识，并以此为起点，在未来进行更加深入的了解。

这一讲，就让我们来一起了解下并发编程的主角，线程，以及我们在围绕它进行多线程开发时可能遇到的一些常见问题。

进程 vs 线程

相信你对“进程”这个概念应该比较熟悉，那么线程与进程有哪些区别呢？我们来一起看一看。

默认情况下，操作系统会为每一个运行中的程序创建一个相应的进程，以作为它的运行实例。而进程中则包含与该程序有关的一系列运行时信息，比如 VAS、进程 ID、处理器上下文（如通用目的寄存器与指令寄存器中的值）、进程状态，以及操作系统分配给该进程的相关资源等。这些信息被统一存放在内核提供的，名为“进程控制块（PCB）”的数据结构中。

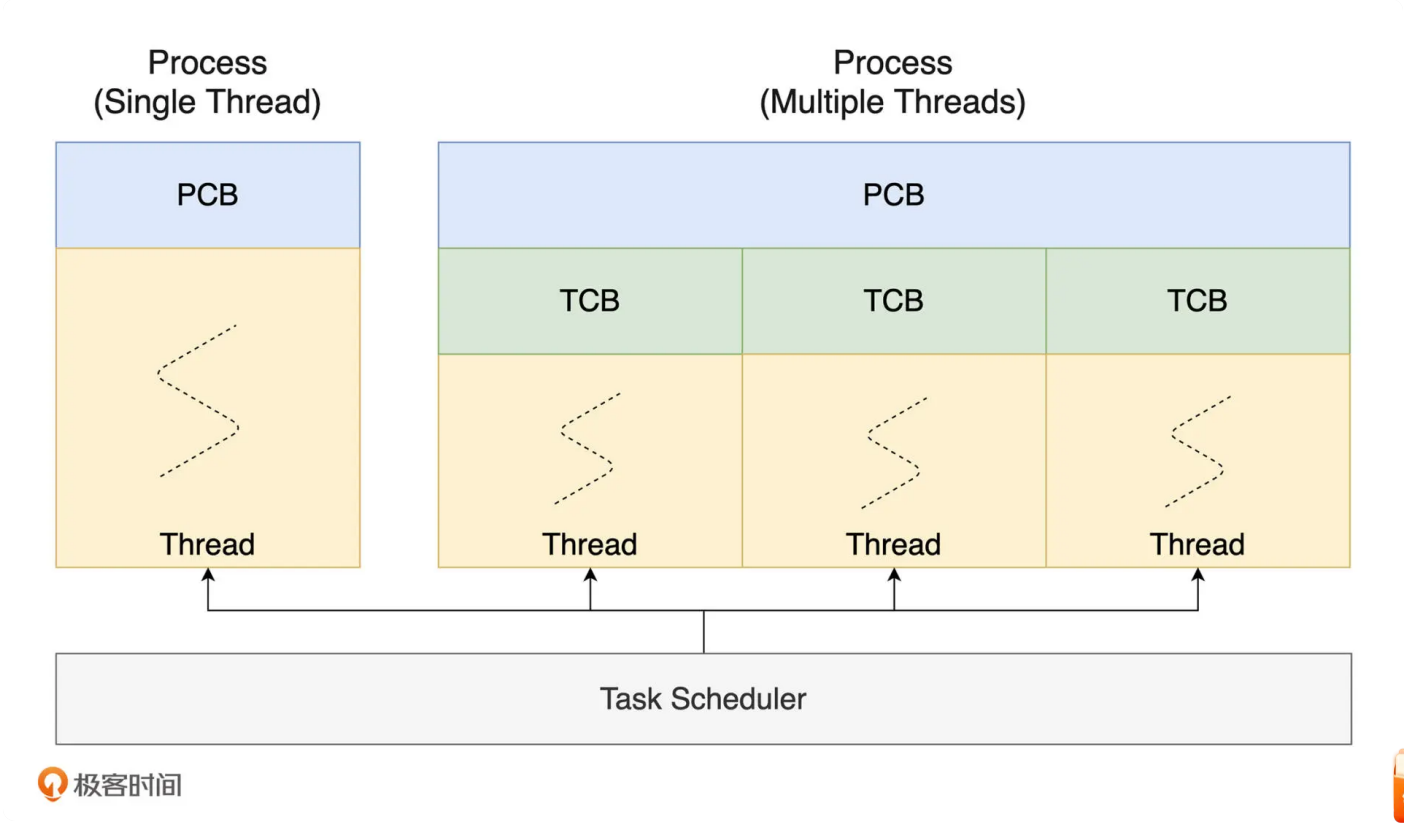
现代 CPU 通常会采用“抢占式”调度算法，来进行多个进程之间的任务切换过程。比如，当某个进程的执行时间超过一定阈值，或开始等待 IO 响应，或所在系统出现硬件中断等情况时，内核任务调度器可能会将该进程挂起，并将 CPU 资源“转移”给其他进程使用。通过这种方式，即使是在单核 CPU 上，操作系统也能够实现用户可观测的多任务处理过程。当然，具体的调度算法还会考虑进程的优先级权重、历史执行情况等因素。调度器只有在经过“综合评定”后，才会作出选择。

而相较于进程，线程则为程序提供了更细粒度的运行单元。对于大多数传统操作系统实现来说，在默认情况下，每一个进程内部都会存在至少一个线程。其中，进程负责划分不同程序所

享有资源的边界；而线程则在共享程序运行资源的情况下，负责程序某个子任务的具体执行过程。

此时，任务调度器也将以线程作为最小的调度实体，来更加精细地组织程序的运行。因此，通过增加进程中线程的个数，我们便能够将程序需要完成的任务进行更具体的划分（比如将 IO 相关操作单独分配给一个线程来执行），并同时进一步利用多核 CPU 的计算资源。

同进程类似，每一个线程所具有的不同状态信息被保存在内核中名为“线程控制块（TCB）”的数据结构中。其中包含有与特定线程运行紧密相关的处理器上下文、线程 ID、所属进程，以及状态信息，等等。可以看到的是，PCB 与 TCB 两者内部所包含的内容会有一定重合。这是由于在某些操作系统上，单进程（单线程）应用的运行可能会直接使用 PCB，来同时保存程序的资源描述信息与执行状态信息。而当运行多线程应用时，才会配合使用 PCB 与 TCB。这里，你可以通过下图来直观地对比进程与线程之间的区别和联系。当然，具体的实现方式并不唯一。



聊完了基本的理论知识，接下来就让我们看看如何在 C 代码中创建线程。

线程的基本控制

借助 threads.h 头文件提供的接口，我们可以实现对线程的创建、等待、阻塞、分离，以及重新调度等操作。来看下面这段示例代码：

复制代码

```
#include <threads.h>
#include <stdio.h>
int run(void *arg) {
  thrd_t id = thrd_current();  // 返回该函数运行所在线程的标识符；
  printf((const char*) arg, id);
  return thrd_success;
}
int main(void) {
#ifndef __STDC_NO_THREADS__
  thrd_t thread;
  int result;
  // 创建一个线程；
  thrd_create(&thread, run, "Hello C11 thread with id: %lu.\n");
  if (thrd_join(thread, &result) == thrd_success) {
    // 等待其他线程退出；
    printf("Thread returns %d at the end.\n", result);
  }
#endif
  return 0;
}
```

可以看到，在 main 函数的开头，我们首先通过判断宏 **STDC_NO_THREADS** 是否存在，来决定是否“启用”多线程相关的代码。C11 标准中规定，若编译器实现不支持并发编程相关接口，则需要定义该宏。

然后，在条件编译指令 `ifndef` 的内部，我们定义了类型为 `thrd_t` 的变量 thread。该类型为由具体实现定义的，用于标识某个线程的唯一对象。接下来，在代码的第 13 行，我们通过 thrd_create 函数创建了一个线程。这里，传入该函数的第二个参数为一个 `thrd_start_t` 类型的函数指针，该指针所指向函数将会在这个新线程内被执行，其接收到的参数为 thrd_create 函数调用时传入的第三个参数。

当新线程创建完毕后，其执行过程便与 main 函数所在线程发生了“分离”。因此，为了防止 main 函数执行结束前（即整个程序退出前），新线程内的代码还没有执行完成。这里，我们便需要通过 thrd_join 函数，来让 main 函数所在线程“等待”新线程的执行结束。当然，除此之外，如果仅是为了能让新线程顺利执行完毕，我们也可以用 thrd_exit 函数来达到这个目的。这里你可以思考下具体应该怎样做，并在评论区留下你的答案。

接下来，函数 run 在新线程中被执行。通过函数 thrd_current，我们可以得到当前代码执行所在线程的唯一标识符。然后，调用 printf 函数，我们将该标识符的值打印了出来。至此，新线程执行完毕并终止，执行流程又回到 main 函数所在线程中。

实际上，除了我们在上面例子中使用到的以 “thrd_” 开头的函数外，C11 还为我们提供了一些其他的线程控制函数。这里，我将它们整理到了下面的表格中，供你参考：

函数名	功能描述
thrd_equal	检查两个 thrd_t 对象是否代表同一个线程
thrd_sleep	持续阻塞当前线程执行直至经过给定时间，或收到特定信号
thrd_yield	通知操作系统重新调度当前线程的执行
thrd_exit	终止当前线程的执行
thrd_detach	从当前环境分离一个指定线程

随着多线程应用的功能变得逐渐复杂，共享变量可能会被多个线程同时访问。并且，不同线程可能会以不同的先后顺序，来执行程序中的同一段代码。除此之外，现代 CPU 采用的特殊指令处理方式，使得程序的实际执行流程与对应的汇编代码可能也不完全一致。而上述这三个因素，都确实会在某些情况下影响多线程应用的正常执行。

下面，我们就来了解一下与上述这三种情况相对应的三个概念，即：数据竞争、竞态条件，以及指令重排。

数据竞争

从定义上来讲，数据竞争（Data Race）是指在一个多线程环境中，有两个及以上的线程在同一时间对同一块内存中的数据进行了非原子操作，且其中至少有一个是写操作。在这种情况下，该数据值的最终状态可能与程序语义上希望表达的计算结果不一致。来看下面这个例子：

```c
#include <threads.h>
#include <stdio.h>
#define THREAD_COUNT 20
#define THREAD_LOOP 100000000
long counter = 0;  // 全局变量，用来记录线程的累加值；
int run(void* data) {
  for (int i = 0; i < THREAD_LOOP; i++)
    counter++;  // 在线程中递增全局变量的值；
  printf("Thread %d terminates.\n", *((int*) data));
  return thrd_success;
}
int main(void) {
#ifndef __STDC_NO_THREADS__
  int ids[THREAD_COUNT];  // 用于存放线程序号的数组；
  thrd_t threads[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; i++) {
    ids[i] = i + 1;
    thrd_create(&threads[i], run, ids + i);  // 创建 THREAD_COUNT 个线程；
  }
  for (int i = 0; i < THREAD_COUNT; i++)
    thrd_join(threads[i], NULL);  // 让当前线程等待其他线程执行完毕；
  printf("Counter value is: %ld.\n", counter);  // 输出 counter 变量最终结果；
#endif
  return 0;
}
```

在这段代码中，我们在 main 函数内创建了 20 个线程（由宏常量 THREAD_COUNT 指定），并让这些线程同时对全局变量 counter 进行值递增操作。由于 counter 的初始值被设置为 0，因此，如果代码按照我们的预期执行，20 个线程分别对该全局变量递增 1 亿次，程序在退出前打印的 counter 变量值应该为 20 亿。但实际情况可能并非如此，下面我们具体看下。

在非优化情况下编译并多次运行这段代码，你会发现程序打印出的 counter 变量值并不稳定。在某些情况下，这个值是准确的，而某些情况下却小于这个数字。这个问题便是由于多线程模型下的数据竞争引起的。

对于上面这个例子来说，编译器可能会将 run 函数内的 counter 变量自增语句，编译为如下所示的几条机器指令的组合：

复制代码

```
mov eax, DWORD PTR counter[rip]
add eax, 1
mov DWORD PTR counter[rip], eax
```

在这种情况下，当多个线程在 CPU 的调度下交错运行时，便可能会发生这样一种情况：某个线程刚刚执行完上述代码的第一条指令，将变量 counter 的值暂存在了寄存器 rax 中。此时，操作系统开始调度线程，将当前线程挂起，并开始执行另一个线程的代码。

新的线程在执行递增时，由于需要再次从 counter 所在的原内存地址中读入数据，因此，该值与上一个线程读取到的数据是相同的。而这便会导致这两个线程在递增后得到的结果值也完全相同，两个线程对 counter 变量的两次递增过程仅使得它的原值增加了 1。

不仅如此，哪怕编译器在优化情况下，可以将上述递增语句实现为仅一条汇编指令，数据竞争的问题仍可能会存在。

比如，编译器将该递增操作实现为机器指令 `add DWORD PTR counter[rip], 1`（这里使用 RIP-relative 寻址）。现代 x86-64 处理器在处理这条 CISC 风格的机器指令时，可能会将其拆分为对应的三种不同“微指令（uOp）”：LOAD、ADD、STORE。其中，LOAD 指令会首先从给定内存地址处读出当前的数据值；ADD 指令则会根据用户传入的立即数参数，来计算出更新后的数据值；最后，STORE 指令会将这个结果数据值更新到对应的内存中。同之前多条机器指令的实现类似，这些微指令在操作系统的线程调度下，也可能存在着交替执行的过程，因此也有着产生数据竞争的风险。

竞态条件

竞态条件（Race Condition）是指由于程序中某些事件的发生时机与顺序不一致，从而影响程序运行正确性的一种缺陷。在某些情况下，数据竞争的存在可能会导致竞态条件的出现，但两者的出现实际上并没有太多联系（有部分人认为数据竞争是竞态条件的一种，但也有人持反对意见）。

不同于数据竞争的是，对程序中竞态条件的判断可能是非常困难的。竞态条件并没有可以精确到具体操作和行为上的定义，因此，它是否产生完全取决于程序的具体设计，以及是否存在可能影响程序运行的外部非确定性变化。

比如，我们来看下面这段仅含有竞态条件，但并没有数据竞争的示例代码：


```
#include <threads.h>
#include <stdio.h>
#include <stdatomic.h>
```

```c
#include <stdlib.h>
#include <time.h>
#define THREAD_COUNT 10
atomic_int accountA = 100000000;  // 转出账户初始金额；
atomic_int accountB = 0;  // 转入账户初始金额；
int run(void* v) {
  int _amount = *((int*) v);  // 获得当前线程的转移金额；
  for(;;) {
    // 首先判断转出账户金额是否足够，不够则直接退出；
    if (accountA < _amount) return thrd_error;
    atomic_fetch_add(&accountB, _amount);  // 将金额累加到转入账户；
    atomic_fetch_sub(&accountA, _amount);  // 将金额从转出账户中扣除；
  }
}
int main(void) {
#if !defined(__STDC_NO_THREADS__) && !defined(__STDC_NO_ATOMICS__)
  thrd_t threads[THREAD_COUNT];
  srand(time(NULL));
  for (int i = 0; i < THREAD_COUNT; i++) {
    int amount = rand() % 50;  // 为每一个线程生成一个随机转移金额；
    thrd_create(&threads[i], run, &amount);
  }
  for (int i = 0; i < THREAD_COUNT; i++)
    thrd_join(threads[i], NULL);
  printf("A: %d\nB: %d", accountA, accountB);
#endif
  return 0;
}
```

在这段代码中，我们以简化的方式实现了一个基本的金融场景，即账户 A 向账户 B 进行多次转账，且每次转账的金额都并不固定。在 main 函数的第 22~25 行，我们通过创建多个线程的方式来模拟两个账户之间分批、多次的财产转移过程。其中，每个线程会使用不同的固定份额来进行转账过程，而随机数 amount 便表示了这个额度。

线程开始运行后，在代码第 10 行的 run 函数内，我们通过一个线程本地变量 _amount 来存放传入的、当前线程需要使用的固定份额。接下来，在一个无限循环中，线程会通过以下三个步骤，来完成两个账户之间的转账过程：

1. 判断账户 A 的剩余资金是否足够进行转账。若否，则直接退出；
2. 将转账的金额累加到账户 B 中；
3. 将转账的金额从账户 A 中扣除。

而当所有线程都退出时，则表示当前账户 A 中的剩余金额，无法再满足任何一个线程以它的固定金额为单位进行转账。最后，通过 printf 函数，我们将两个账户内的剩余资金打印了出来。按照我们对程序的理解，此时，两个账户中的金额应该都大于 0，且两者之和为 1 亿。

可以看到，程序的整个执行流程十分简单。不仅如此，我们还使用了原子操作，来保证程序对账户变量 accountA 与 accountB 的修改过程是不会产生数据竞争的。有关原子操作的更多内容，我会在下一讲中为你介绍，这里你可以先对它的用法有一个基本了解。

到这里，程序的逻辑看起来没有任何问题。但当我们真正运行它时，却可能会得到如下所示的输出结果：

```
1 A: -46
2 B: 100000046
```

可以看到，账户 A 的剩余金额变为了负数，程序运行出现了异常。如果仔细分析上述代码的执行逻辑，你会发现多个线程在对账户变量进行修改时，虽然没有数据竞争，但程序的不恰当设计导致其存在着竞态条件。

当多个线程在同时执行 run 函数内的逻辑时，操作系统中的任务调度器可能会在任意时刻暂停某个线程的执行，转而去运行另一个线程。因此，便可能出现这样一种情况：某个线程以原子形式，执行了代码的第 14 行语句，将金额累加到账户 A。而此时，调度器将执行流程转移给另一个线程。该线程在上一个线程还没有完成对账户 B 的扣减操作前，便直接使用未同步的值参与了下一次的转账操作。

因此，在这种情况下，程序的正确性实际上依赖于各个线程之间，按照一定顺序的执行过程。那么，应该如何修复这个程序呢？学完下一讲后，你就可以结合这两讲的知识，尝试解决这个问题了。

最后，我们再来看另一个与并发编程密切相关的话题，指令重排。

指令重排

现代编译器和处理器通常会采用名为“指令重排”（乱序执行的一种）的技术来进一步提升程序的运行效率。这种技术会在尽量不影响程序可观测执行结果的情况下，对生成的机器指令，或

它们的实际执行顺序进行适当的重新排序。对于编译器来说，其表象是源代码中语句的出现顺序，与对应汇编代码的实现顺序不一致。而对于处理器来说，则是**程序在真正执行时产生副作用的顺序（比如变量赋值），与汇编代码中指令的出现顺序不一致**。

对多线程应用来说，即使编译器可以从静态分析的视角，来确保汇编指令的重排不影响程序的可观测执行结果，但当多个线程被调度到不同的物理 CPU 上执行时，不同 CPU 之间一般无法共享对应线程指令在执行时的重排信息。因此，当线程之间存在数据依赖关系时，程序的运行时正确性可能会受到影响。比如，我们来看下面这个例子：

复制代码

```c
#include <threads.h>
#include <stdio.h>
#include <stdatomic.h>
#if !defined(__STDC_NO_ATOMICS__)
atomic_int x = 0, y = 0;
#endif
int run(void* v) {
  x = 10;
  y = 20;  // ! 变量 y 的值可能被优先更新!
}
int observe(void* v) {
  while(y != 20) ;  // 忙等待;
  printf("%d", x);  // 只在 x 被更新后打印;
}
int main(void) {
#if !defined(__STDC_NO_THREADS__)
  thrd_t threadA, threadB;
  thrd_create(&threadA, run, NULL);
  thrd_create(&threadB, observe, NULL);
  thrd_join(threadA, NULL);
  thrd_join(threadB, NULL);
#endif
  return 0;
}
```

在这段代码中，我们在 main 函数内生成了两个线程，分别对应于函数 run 和 observe。其中，observe 线程在执行时会通过“忙等待”的方式，不断查询变量 y 的状态，并在发现其值变更为 20 后，再继续执行接下来的 printf 函数。而 run 线程在执行时，仅会简单地将变量 x 更新为值 10，然后再将变量 y 的值更新为 20。这是我们通过查看上述 C 代码得出的结论。

但由于指令重排的存在，run 函数在程序实际执行时，其内部对变量 x 与 y 的值变更过程，可能与我们在 C 代码中观察到的顺序并不一致。在某些情况下，变量 y 的值可能会被优先更

新。而如果此时 observe 线程被重新调度，则 printf 语句便会打印出并非我们所期望的值。

为此，C 语言为我们提供了相应的被称为“内存顺序（Memory Order）”的一系列枚举值。通过配合特定的库函数一起使用，我们能够明确规定编译器及处理器应该如何对某段代码的指令重排进行约束。下一讲，在“使用原子操作”小节中，我们还会回顾上面的这个例子。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要介绍了与 C 并发编程有关的一些基本概念。当然，这些概念本身都较为通用，它们也可以应用在其他编程语言的环境中。

我们首先对比了进程与线程两者之间的不同。其中，进程主要划分了运行程序所享有的资源边界；而线程则在共享进程资源的情况下，独立负责不同子任务的执行流程。通过使用多线程，程序可以利用多核 CPU 的计算资源，做到真正的任务并行。

接下来，我介绍了如何在 C 代码中创建线程。C 标准将与线程控制相关的接口整合在了名为 threads.h 的头文件中。借助 thrd_create、thrd_join 等函数，我们可以实现对线程的创建、等待、阻塞、分离等一系列操作。

最后，多线程应用的正确执行离不开我们对程序的合理设计。我还讲解了由数据竞争、静态条件，以及指令重排等因素引起的潜在问题。掌握了这些知识，再合理使用我将在下一讲中介绍的其他 C11 并发编程特性，你就能在一定程度上解决和避免这些问题。

思考题

什么类型的应用更适合使用多线程技术来提升其运行时性能？你可以先试着查找资料，并在评论区分享你的思考。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　12 | 标准库：非本地跳转与可变参数是怎样实现的？

下一篇　14 | 标准库：如何使用互斥量等技术协调线程运行？

精选留言 (4)

=

2022-01-12

1. IO密集型的应用。因为CPU等待IO完成的时间很多，在等待的时间内，多线程可以让其他应用继续执行

2. 有多核CPU时的计算密集型应用。因为可以利用多核CPU并行的优势，来快速完成计算

作者回复: 回答的很棒！

2

赵岩松

2022-01-12

对于PCB我一直有一个疑惑，在我的理解中这个概念是操作系统统一的一个抽象概念，同时据我所知Linux中存在两个进程相关的结构：`thread_info`和`task_struct`，那么PCB和这两个结构有没有对应关系呢？至于TCB是我今天第一次听到的一个概念，同样的这个概念与前面提到的两个Linux结构有对应关系吗？

作者回复: 是的，其实 PCB 与 TCB 就是针对进程和线程的两个不同的抽象概念，具体实现是可以多种多样的。简单来讲，Linux 在实现中没有为进程和线程单独设置用于存放相关信息的数据结构，而是统一使用 task_struct 存储通用的任务信息，使用 thread_info 存放平台相关的信息。区分是不是线程就看不同 task_struct 对象之间有没有共享资源。即你可以认为在 Linux 中只有进程（PCB），没有独立的线程（TCB）。而相较于 Windows 和 Sun Solaris 等明确区分进程与线程的操作系统实现，Linux 的实现相对更加优雅。

— 参考自《Linux Kernel Development 3rd Edition》，基于 2.6 版本内核。

3

焚心以火

2022-02-09

大家有遇到 fatal error: 'threads.h' file not found 是怎么解决的呢？

作者回复: 大多数情况是由于编译器不支持 C11 的多线程特性，你可以通过检测宏常量 __STDC_NO_THREADS__ 来判断当前编译器是否支持 threads.h，然后仅在支持的情况下再 include。

ppm

2022-01-12

请问一个进程里面长时间有多个线程 好不好

作者回复: 可以再描述的具体点？

共 4 条评论>