下载APP

05 | 栈的魔法：从栈切换的角度理解进程和协程

2021-11-03 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 20:25 大小 18.70M

你好，我是海纳。

上一节课，我们了解到函数在执行的时候，就会在栈上创建栈帧，那么函数执行的上下文都将保存在栈帧里。今天，我们就再来进一步分析，栈切换在计算机系统设计中所发挥的重要作用。

几乎所有的程序员都会遇到并发程序。因为多进程或者多线程程序可以并发执行，充分利用多 CPU 多核的计算资源来完成任务，会大大提升应用程序的性能。

所以，我相信你在工作中也遇到过多线程程序，但不知道你是否考虑过进程和线程是如何切换的呢？很多文章都介绍了，操作系统为了避免频繁进入内核态，会把很多工作都尽量放在用户态。那么你有没有仔细思考过内核态、用户态到底意味着什么呢？

要回答上面的问题，我们就要理解这些概念背后最重要的一个步骤：对执行单元的上下文环境进行切换。它就是由栈这个核心数据结构支撑的，这也是我们今天学习的重点内容。

通过今天的学习，你将掌握协程的基本知识，这样，你在 C++ 中使用各种协程库，或者在 Lua、Go 等语言中使用原生协程的时候，就能理解它们背后发生了什么，也可以帮你写出正确的 IO 程序。你还将深入理解操作系统用户态和内核态，这样，你在做架构的时候，就能正确评估操作系统进入内核态的开销是多少。

在讲解执行单元的切换与栈的关系之前，我们先来给出它的准确定义。

什么是执行单元

执行单元是指 CPU 调度和分派的基本单位，它是一个 CPU 能正常运行的基本单元。执行单元是可以停下来的，只要能把 CPU 状态（其实就是寄存器的值）全部保存起来，等到这个执行单元再被调度的时候，就把状态恢复过来就行了。我们把这种保存状态，挂起，恢复执行，恢复状态的完整过程，称为执行单元的调度 (Scheduling)。

具体来说，常见的执行单元有进程，线程和协程三种，接下来，我们详细说明这三种执行单元的区别和联系。我们先来比较进程和线程。

理解进程和线程

当运行一个可执行程序的时候，操作系统就会启动一个进程。进程会被操作系统管理和调度，被调度到的进程就可以独占 CPU 了。

CPU 就像是一个可以轮流使用的工作台，多个进程可以在工作台上工作，时间到了就会带着自己的工作离开工作台，换下一个进程上来工作。

进程有自己独立的内存空间和页表，以及文件表等等各种私有资源，如果使用多进程系统，让多个任务并发执行，那么它所占用的资源就会比较多。线程的出现解决了这个问题。

同一个进程中的线程则共享该进程的内存空间，文件表，文件描述符等资源，它与同一个进程的其他线程共享资源分配。除了共享的资源，每个线程也有自己的私有空间，这就是线程的栈。线程在执行函数调用的时候，会在自己的线程栈里创建函数栈帧。

根据上面所说的特点，人们常把进程看做是资源分配的单位，把线程才看成一个具体的执行实体。

由于线程的切换过程和进程的切换过程十分相似，我们这节课就只以进程的切换为重点进行讲解，请你一定要自己查找相关资料，对照进程切换的过程，去理解线程的切换过程。

理解协程

协程是比线程更轻量的执行单元。进程和线程的调度是由操作系统负责的，而协程则是由执行单元相互协商进行调度的，所以它的切换发生在用户态。只有前一个协程主动地执行 yield 函数，让出 CPU 的使用权，下一个协程才能得到调度。

因为程序自己负责协程的调度，所以大多数时候，我们可以让不那么忙的协程少参与调度，从而提升整个程序的吞吐量，而不是像进程那样，没有繁重任务的进程，也有可能被换进来执行。

协程的切换和调度所耗费的资源是最少的，Go 语言把协程和 IO 多路复用结合在一起，提供了非常便捷的 IO 接口，使得协程的概念深入人心。

从操作系统和 Web Server 演进的历史来看，先是多进程系统的出现，然后出现了多线程系统，最后才是协程被大规模使用，这个演进历程背后的逻辑就是执行单元需要越来越轻量，以支持更大的并发总数。

但我们这节课却要先讲协程，这是因为从实现层面来说，协程是最简单的，当你理解了协程的实现原理，再回头学习进程就比较容易了，所以我们先来学习协程的原理。

协程是怎么调度和切换的？

在讲解协程的理论之前，我们先通过一个最简单的协程的例子，来观察协程的运作机制：

复制代码

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

```
2  #include <stdlib.h>
3
4  #define STACK_SIZE 1024
5
6  typedef void(*coro_start)();
7
8  class coroutine {
9  public:
10     long* stack_pointer;
11     char* stack;
12
13     coroutine(coro_start entry) {
14         if (entry == NULL) {
15             stack = NULL;
16             stack_pointer = NULL;
17             return;
18         }
19
20         stack = (char*)malloc(STACK_SIZE);
21         char* base = stack + STACK_SIZE;
22         stack_pointer = (long*) base;
23         stack_pointer -= 1;
24         *stack_pointer = (long) entry;
25         stack_pointer -= 1;
26         *stack_pointer = (long) base;
27     }
28
29     ~coroutine() {
30         if (!stack)
31             return;
32         free(stack);
33         stack = NULL;
34     }
35 };
36
37 coroutine* co_a, * co_b;
38
39 void yield_to(coroutine* old_co, coroutine* co) {
40     __asm__ (
41         "movq %%rsp, %0\n\t"
42         "movq %%rax, %%rsp\n\t"
43         :"=m"(old_co->stack_pointer):"a"(co->stack_pointer):);
44 }
45
46 void start_b() {
47     printf("B");
48     yield_to(co_b, co_a);
49     printf("D");
50     yield_to(co_b, co_a);
51 }
52
53 int main() {
```

```
54     printf("A");
55     co_b = new coroutine(start_b);
56     co_a = new coroutine(NULL);
57     yield_to(co_a, co_b);
58     printf("C");
59     yield_to(co_a, co_b);
60     printf("E\n");
61     delete co_a;
62     delete co_b;
63     return 0;
64 }
```

我们使用 g++ 对这段代码进行编译，注意要使用 O0 进行编译，不能使用更高的优化级别，这是因为更高级别的优化会内联 yield_to 方法，这就使得栈的布局和程序中期望的不相符了。我们先来看看这段代码的运行的结果，如下所示：

```
1 # g++ -g -o co -O0 coroutine.cpp
2 # ./co
3 ABCDE
```

这段代码的神奇之处在于，main 函数在执行到一半的时候，可以停下来去执行 start_b 函数，这和我们通常遇到的函数调用是很不一样的。而这种效果是通过协程达到的。

你可以看到，在 main 函数的执行过程中（即代码的 57 行），CPU 通过执行 yield_to 方法转到另外一个协程。新的协程的入口函数是 start_b，所以，CPU 就转而去执行 start_b，在 start_b 执行到 48 行的时候，还能再通过 yield_to，再回到 main 函数中继续执行。

下面我们来看协程是怎么实现这一点的。

我们调用构造函数 coroutine 创建了两个协程 co_a 和 co_b（即代码的 55、56 行）。其中，co_b 的入口地址是函数 start_b，co_a 没有入口地址。

我们具体来看在 coroutine 里发生了什么。其实在创建这两个协程之前，coroutine 已经申请了一段大小为 1K 的内存作为协程栈，然后让栈底指针 base 指向栈的底部（第 21 行）。因为栈是由上向下增长的，所以，我们又在协程栈上放入了 base 地址和起始地址（第 23~27 行），此时，协程栈内的数据是这样的，如图 1 所示：

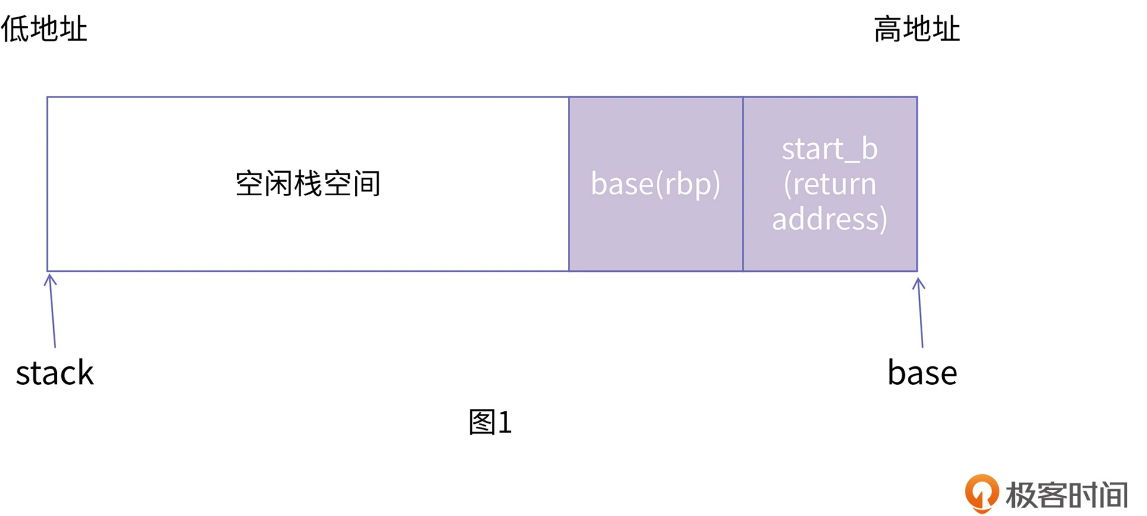


图1

在准备好协程栈以后，就可以调用 yield_to 方法进行协程的切换。在上一节中，我们提到过协程要主动调用 yield 方法将 CPU 的占有权让出来，后面的协程才能执行。所以，协程切换的关键机制就肯定隐藏在 yield_to 方法里。

yield_to 方法具体做了什么事情呢？我们需要通过机器码来进行说明。这里我们使用"objdump -d"命令查看 yield_to 方法经过编译以后的机器码：

复制代码

```
000000000040076d <_Z8yield_toP9coroutineS0_>:
  40076d:       55                      push   %rbp
  40076e:       48 89 e5                mov    %rsp,%rbp
  400771:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
  400775:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  400779:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  40077d:       48 8b 00                mov    (%rax),%rax
  400780:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  400784:       48 89 22                mov    %rsp,(%rdx)
  400787:       48 89 c4                mov    %rax,%rsp
  40078a:       5d                      pop    %rbp
  40078b:       c3                      retq
```

yield_to 中，参数 old_co 指向老协程，co 则指向新的协程，也就是我们要切换过去执行的目标协程。

这段代码的作用是，首先，把当前 rsp 寄存器的值存储到 old_co 的 stack_pointer 属性（第 9 行），并且把新的协程的 stack_pointer 属性更新到 rsp 寄存器（第 10 行），然后，retq 指令将会从栈上取出调用者的地址，并跳转回调用者继续执行（第 12 行，这是上一节课的内容，如果你不熟悉，可以再自行复习一下）。

结合以上分析，我们可以想象在协程示例代码的第 57 行，当调用这一次 yield_to 时，rsp 寄存器刚好就会指向新的协程 co 的栈，接着就会执行"pop rbp"和"retq"这两条指令。这里你需要注意一下，栈的切换，并没有改变指令的执行顺序，因为栈指针存储在 rsp 寄存器中，当前执行到的指令存储在 IP 寄存器中，rsp 的切换并不会导致 IP 寄存器发生变化。

而显然，如图 1 所示，我们刚才精心准备的 base 地址正好就是为了"pop rbp"准备的，而 start_b 则是为了 retq 准备的。执行这次 retq，CPU 就会跳转到 start_b 函数中去运行了。

经过这种切换，系统中会出现两个栈，如图 2 所示：

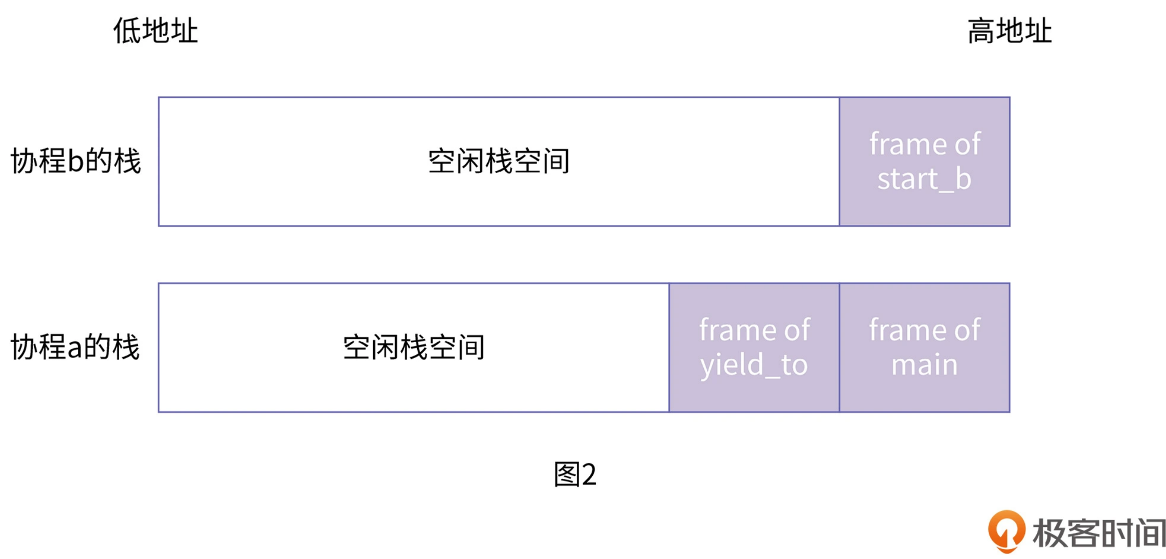


图2

当程序继续执行时，在 start_b 中调用了 yield_to，CPU 又会转移回协程 a 的栈上，这样在执行 retq 时，就会返回到 main 函数里继续运行了。

在这个过程中，我们并没有使用任何操作系统的系统调用，就实现了控制流的转移。也就是说，在同一个线程中，我们真正实现了两个执行单元。这两个执行单元并不像线程那样

是抢占式地运行，而是相互主动协作式执行，所以，这样的执行单元就是协程。我们可以看到，协程的切换全靠本执行单元主动调用 yield_to 来把执行权让渡给其他协程。

每个协程都拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方（上述例子中，保存在 coroutine 对象中），在切回来的时候，恢复先前保存的寄存器上下文和栈。

分析到这里，这个程序对我们而言，已经没有太多秘密了，它所有看上去神奇的地方，不过就是切换了程序运行的栈指针而已。

分析到这里，我们就可以准确地定义协程了。协程是一种轻量级的，用户态的执行单元。相比线程，它占用的内存非常少，在很多实现中（比如 Go 语言）甚至可以做到按需分配栈空间。

它主要有三个特点：

1. 占用的资源更少；
2. 所有的切换和调度都发生在用户态。
3. 它的调度是协商式的，而不是抢占式的。

前两个特点容易理解，我来给你重点解释一下第三个特点。

目前主流语言基本上都选择了多线程作为并发设施，与线程相关的概念是抢占式多任务（Preemptive multitasking），而与协程相关的是协作式多任务。不管是进程还是线程，每次阻塞、切换都需要陷入系统调用 (system call)，先让 CPU 执行操作系统的调度程序，然后再由调度程序决定该哪一个进程 (线程) 继续执行。

由于抢占式调度执行顺序无法确定，我们使用线程时需要非常小心地处理同步问题，而协程完全不存在这个问题。因为协作式的任务调度，是要用户自己来负责任务的让出的。如果一个任务不主动让出，其他任务就不会得到调度。这是协程的一个弱点，但是如果使用得当，这其实是一个可以变得很强大的优点。

你可以尝试将编译优化等级设为 O1，观察 yield_to 函数的机器码的变化，然后就可以理解当栈基址寄存器的保存和恢复如果被优化掉以后，我们准备的那个数据就不再起作用

了。也请你尝试对上述代码进行修改，以适应 O1 优化。

在理解了协程以后，我们再回过头来看进程。

进程是怎么调度和切换的？

进程切换的原理其实与协程切换的原理大致相同，都是将上下文保存在特定的位置，切换到新的进程去执行。所不同的是，操作系统为我们提供了进程的创建、销毁、信号通信等基础设施，这使得程序员可以很方便地创建进程。如果一个进程 a 创建了另外一个进程 b，则称 a 为父进程，b 为子进程。

我先带你通过下面这个例子，直观地感受多进程运行的情况：

复制代码

```
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid;
    if (!(pid = fork())) {
        printf("I am child process\n");
        exit(0);
    }
    else {
        printf("I am father process\n");
        wait(pid);
    }

    return 0;
}
```

编译执行这段代码的结果如下所示：

复制代码

```
# gcc -o p process.c
# ./p
I am father process
I am child process
```

在这个结果里，我们可以看到，在 if 分支和 else 分支中的代码都被运行了。曾经有个笑话说，这个世界上最远的距离，不是你在天涯，我在海角，而是你在 if 里，我在 else 里。由此可见，这个笑话也并不正确，还是要看 if 条件里填的是什么。

在上面的代码中，fork 是一个系统调用，用于创建进程，如果其返回值为 0，则代表当前进程是子进程，如果其返回值不为 0，则代表当前进程是父进程，而这个返回值就是子进程的进程 ID。

我们看到，子进程在打印完一行语句后就调用 exit 退出执行了。父进程在打印完以后，并没有立即退出，而是调用 wait 函数等待子进程退出。由于进程的调度执行是操作系统负责的，具有很大的随机性，所以父进程和子进程谁先退出，我们并不能确定。为了避免子进程变成孤儿进程，我们采用了让父进程等待子进程退出的办法，就是对两个进程进行同步。

其实，这段程序最难理解的是第 6 行，为什么一次 fork 后，会有两种不同的返回值？这是因为 fork 方法本质上在系统里创建了两个栈，这两个栈一个是父进程的，一个是子进程的。创建的时候，子进程完全“继承”了父进程的所有数据，包括栈上的数据。父子进程栈的情况如图 3 所示：

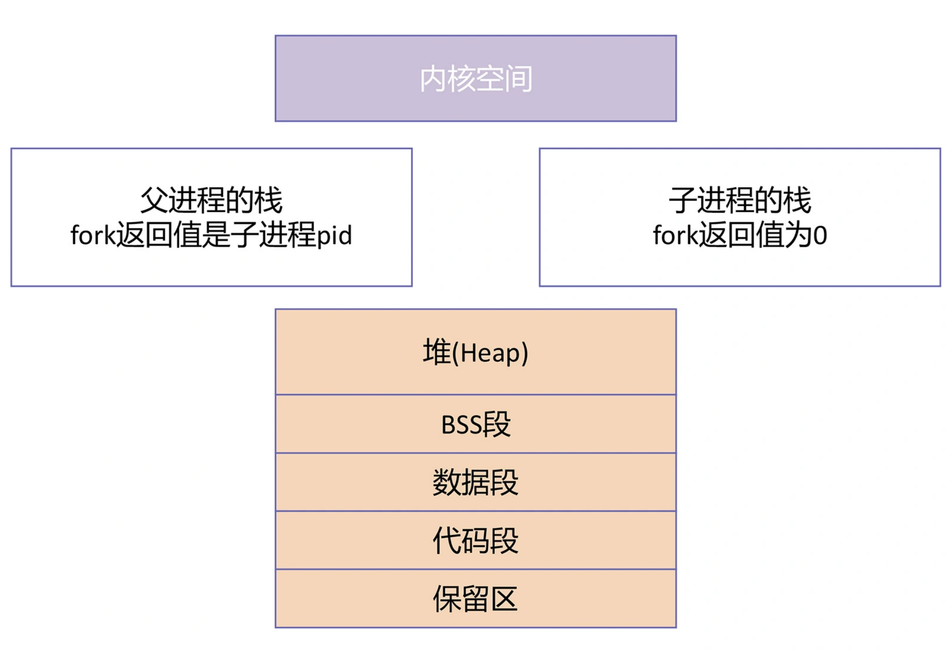


图3

在图 3 里，只要有一个进程对栈进行修改，栈就会复制一份，然后父子进程各自持有一份。图中的黄色部分也是进程共用的，如果有一个进程修改它，也会复制一份副本，这种机制叫做写时复制。

接着，操作系统就会接管两个进程的调度。当父进程得到调度时，父进程的栈上是 fork 函数的 frame，当 CPU 执行 fork 的 ret 语句时，返回值就是子进程的 ID。

而当子进程得到调度时，rsp 这个栈指针就将会指向子进程的栈，子进程的栈上也同样是 fork 函数的 frame，它也会执行一次 fork 的 ret 语句，其返回值是 0。

所以第 6 行虽然是同一个变量 pid，但实际上，它在子进程的 main 函数的栈帧里有一个副本，在父进程的栈帧里也有一个副本。从 fork 开始，父进程和子进程就已经分道扬镳了。你可以将进程栈的切换与协程栈的切换对比着进行学习。

我们通过一个例子展示了进程是如何创建的，并且分析了进程创建背后栈的变化过程。你可以看到，进程做为一种执行单元，它的切换还是要依赖于栈切换这个核心机制。

关于 fork 的更多的细节，我们将在第 10 课再加以分析。在这节课，将进程的栈类比于协程栈已经足够了。

用户态和内核态是怎么切换的？

在第二节课里，我们讲解了中断描述符表，并且用系统调用 write 这个例子，来展示如何通过软中断进入内核态。实际上，内核态和用户态的切换也依赖栈的切换。因为在第二节课里，我们还没有讲到栈，所以在讲到用户态切换内核态的时候，并没有涉及到栈的切换，现在，我们补上用户态和内核态切换的最后一块拼图。

操作系统内核在运行的时候，肯定也是需要栈的，这个栈称为内核栈，它与用户应用程序使用的用户态栈是不同的。只有高权限的内核代码才能访问它。而内核态与用户态的相互切换，其中最重要的一个步骤就是两个栈的切换。

中断发生时，CPU 根据需要跳转的特权级，去一个特定的结构中（不同的 CPU 会有所不同，比如 i386 就存在 TSS 中，但不管是什么 CPU，一定会有一个类似的结构），取得目标特权级所对应的 stack 段选择子和栈顶指针，并分别送入 ss 寄存器和 rsp 寄存器，这就完成了一次栈的切换。

然后，IP 寄存器跳入中断服务程序开始执行，中断服务程序会把当前 CPU 中的所有寄存器，也就是程序的上下文都保存到栈上，这就意味着用户态的 CPU 状态其实是由中断服务程序在系统栈上进行维护的。如图 4 所示：

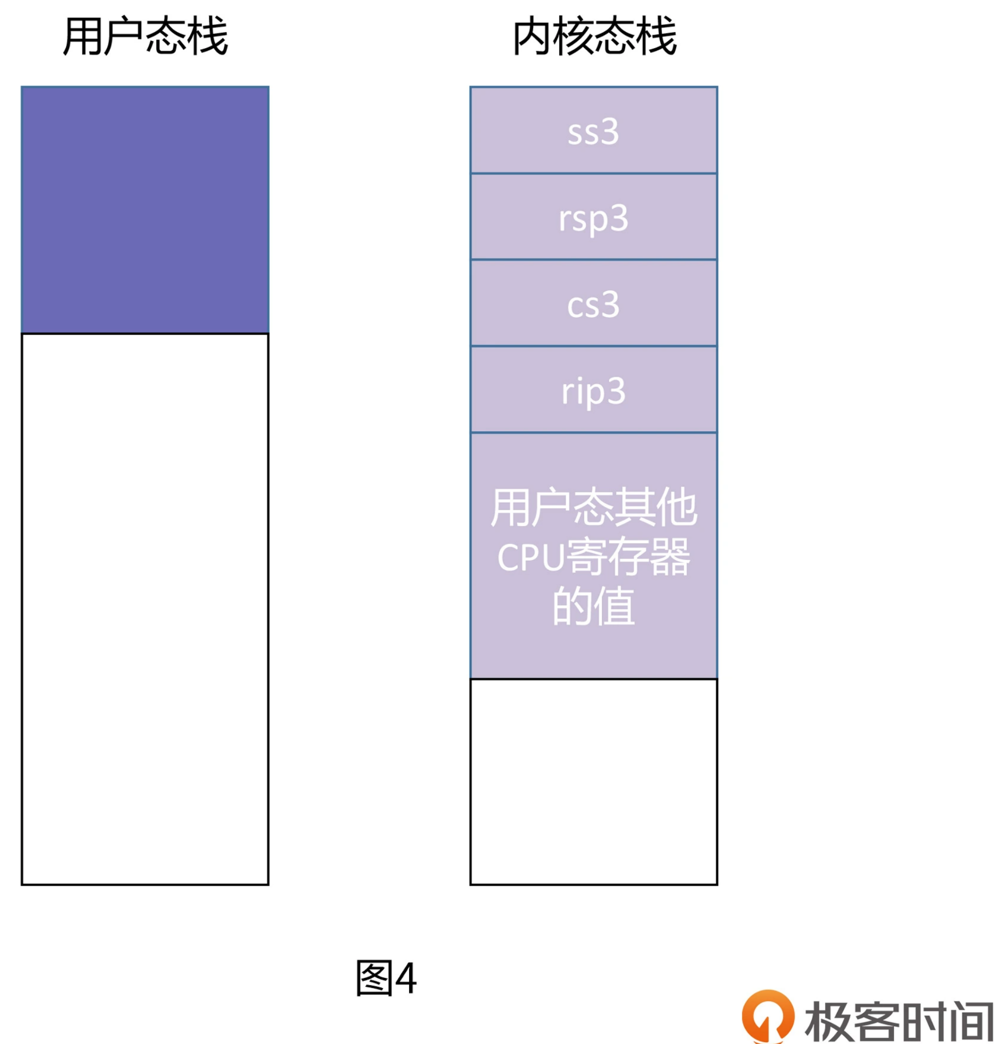


图4

一般来说，当程序因为 call 指令或者 int 指令进行跳转的时候，只需要把下一条指令的地址放到栈上，供被调用者执行 ret 指令使用，这样可以便于返回到调用函数中继续执行。但图 4 中的内核态栈里有一点特殊之处，就是 CPU 自动地将用户态栈的段选择子 ss3，和栈顶指针 rsp3 都放到内核态栈里了。这里的数字 3 代表了 CPU 特权级，内核态是 0，用户态是 3。

当中断结束时，中断服务程序会从内核栈里将 CPU 寄存器的值全部恢复，最后再执行"iret"指令（注意不是 ret，而是 iret，这表示是从中断服务程序中返回）。而 iret 指令就会将 ss3/rsp3 都弹出栈，并且将这个值分别送到 ss 和 rsp 寄存器中。这样就完成了从

内核栈到用户栈的一次切换。同时，内核栈的 ss0 和 rsp0 也会被保存到前文所说的一个特定的结构中，以供下次切换时使用。

总结

这节课我们举例说明了进程，线程和协程的基本概念，并对它们的调度做了简单的说明。然后介绍了服务端编程模型从多进程向协程演进的历程。

接着，我们重点介绍了栈切换的整个过程。**栈切换的核心就是栈指针 rsp 寄存器的切换，只要我们想办法把 rsp 切换了就相当于换了执行单元的上下文环境**。这一节课所有的讲解都可以归到这条线索上。

我们又用了协程切换，进程栈的写时复制和切换，以及用户态和内核态的切换这三个例子来说明举例说明栈的切换所发挥的重要作用。

通过两节课的学习，我们对进程中的栈空间相关的知识进行一次比较深入的梳理。从中我们可以得到一个结论：栈往往和执行单元是一对一的关系，栈的活跃就代表着它所对应的执行单元的活跃。栈上的数据非常敏感，一旦被攻击，往往会造成巨大的破坏。

在第三节课里，我们学习了堆空间的管理方式，这两节课又学习了栈空间的运行机制，这两部分内容都是程序运行时所要操作的内存。在这之后，我们将目光转移到程序的汇编代码，研究一下程序的静态数据是如何组织和划分的。

思考题

我们这节课讲到了协程和进程的栈切换，但没有讲线程的栈切换。请你思考，线程的栈切换是更类似协程那种提前创建好的方式，还是更类似于进程那种按需写时复制？为什么？

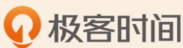
吊打面试官

- 谈谈你对协程的理解?

由于JS和Python中存在Generator机制，Lua和Go语言中有标准的stackful协程实现。所以协程成为面试中的一大高频热点。

这道题要从以下几个方面进行回答，第一，要明确协程是一种轻量级的执行单元；第二，要从协程的调度上入手讲清楚协程是协作式调度，而不像进程和线程是抢占式调度；第三，重点强调协程的切换其本质是依赖栈的切换，每个协程的上下文都保存在自己的栈上；最后是可选的加分项，可以结合IO多路复用的接口讲如何使用协程来实现高性能的同步IO。

高频面试真题



欢迎你在留言区分享你的想法和收获，我在留言区等你。如果这节课帮到了你，也欢迎你把这节课分享给自己的朋友。我们下一讲再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 深入理解栈：从CPU和函数的视角看栈的管理

下一篇 06 | 静态链接：变量与内存地址是如何映射的？

11.11 全年底价

VIP 年卡限定 3 折

畅学 200 门课程 & 新课上线即解锁

超值拿下 ¥999

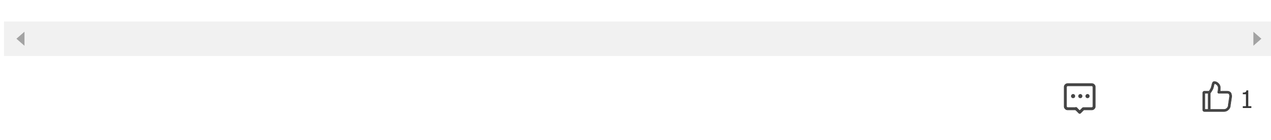
精选留言 (9)

鸮

2021-11-03

补充一下：关于coroutine,在优化O1下，因为没有基地址的入栈出栈操作，所以 代码只用删除coroutine下的基地值入栈操作：stack_pointer -= 1; *stack_pointer = (long) base; 就可以了

作者回复: Good

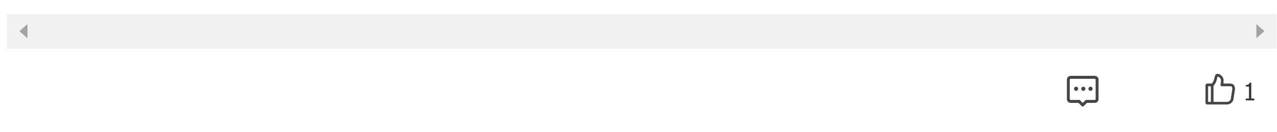


linker

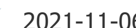
2021-11-03

思考题：线程的用户栈是pthread函数提前创建的吗？

作者回复: 回答正确！写时复制只能发生在进程之间。同一个进程的线程因为共享资源，只能提前分配。



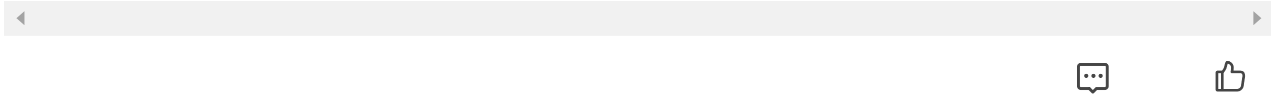
Geek_fd760d

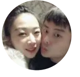

2021-11-06

对于协程的案例中，如果start_b函数中并没有切换回协程a的代码，那么这个函数执行完后还能进入主函数吗？具体会发生什么？

展开

作者回复: 不能，在执行startb的ret指令的时候，就会发现rbp和rip寄存器的值都不正确了。大概率会出现SIGSEGV或者SIGBUS错误。



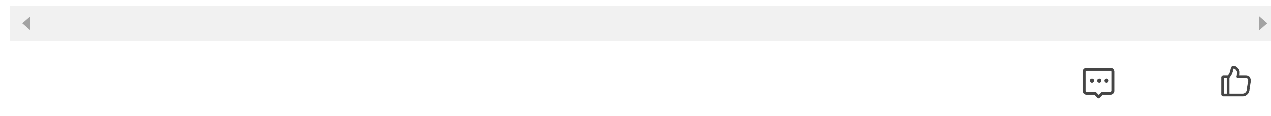


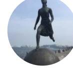
鸮
2021-11-03

对于学java的我看懂这段c++代码真不容易啊，好歹看懂了 。最难懂的应该是 协程a了，一开始一直觉得参数NULL，那个stack_pointer就是个null，后来查查资料，指针变量在声明的时候已经分配内存地址了，所以a的stack_pointer就是内存中的一块地址，只是它指向的是NULL，本身还是一个内存地址。它的作用其实就是为了保存mian函数的rsp，所以才不用赋值（或者说给指定的地址）。...

展开

作者回复: 哈哈哈，说得很有趣也很正确



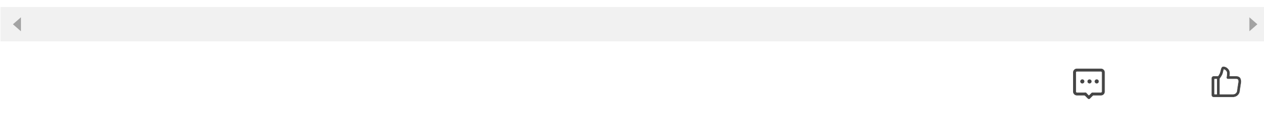


张贺
2021-11-03

每个线程都可以分配一个内核栈吗？

展开

作者回复: Linux系统是这样的。这是因为Linux系统没有很好地支持线程，而是直接利用了进程的结构。当然，Linux的多线程支持也在不断地完善中，未来会是怎么样，我们拭目以待。

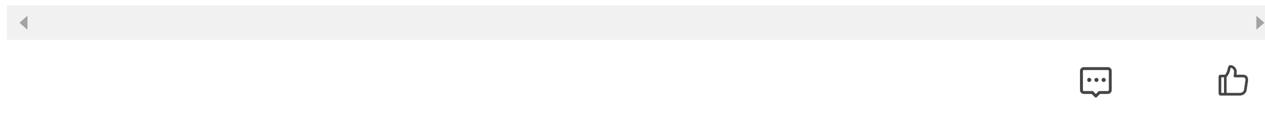


张贺
2021-11-03

用户态内核态切换的时候，线程切换了吗？

展开

作者回复: 没有。当前线程的用户态的状态还会保存在内核态栈上。



独孤

2021-11-03

下面这段代码中co->stack_pointer的值为什么会存到寄存器%rax中？这是约定的吗？

```
void yield_to(coroutine* old_co, coroutine* co)
{ __asm__ ( "movq %%rsp, %0\n\t"
            "movq %%rax, %%rsp\n\t"
            :"=m"(old_co->stack_pointer):"a"(co->stack_pointer):);...
```

展开

作者回复: 因为我们用了"a"(co->stack_pointer)来约束它只能送到rax寄存器里。"a"代表了rax寄存器。关于内嵌汇编，我们在导学的第三节课里有讲，可供参考。

pedro

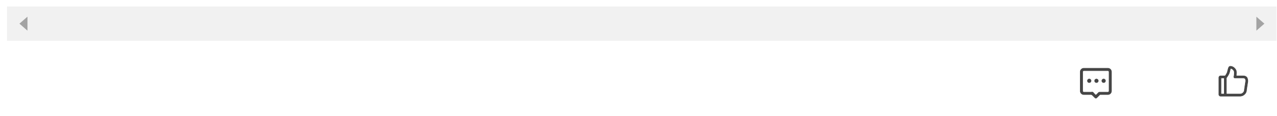
2021-11-03

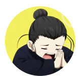
fork 系统调用，只会新建一个子进程调用栈，并向其拷贝父进程调用栈数据，所以父子进程保存了同样的寄存器数据，因此造成了 fork 调用一次，而返回两次的现象。

由于 fork 本质会将父进程拷贝一份，作为子进程，这样就会有大量的拷贝工作，fork 会很慢，同时子进程可能不需要这么多的资源，会造成资源浪费，因此主流操作系统，比如 li...

展开

作者回复: 赞！很准确了。我们在第10节课会讲页中断，那时候就会更加深一层理解。





费城的二鹏

2021-11-03

“其实，这段程序最难理解的是第 6 行，为什么一次 fork 后，会有两种不同的返回值？这是因为 fork 方法本质上在系统里创建了两个栈，这两个栈一个是父进程的，一个是子进程的。”

fork 后，父进程与子进程会有单独的栈与公用栈，我想了解下这个地方的数据结构是怎...

展开

作者回复: 写时复制是按页进行复制的。在创建子进程的时候呢，子进程只是把页表复制一份，所以子进程和父进程的虚拟地址一开始是指向相同的物理地址的。只有当有一个进程在写的时候才会复制一页出来。你要结合第一课想一下，虚拟空间是连续的，但物理空间不必是连续的，所以我在背后悄悄地把映射到的物理地址替换掉，你其实是不知道的。再想一下？