



下载APP



13 | 物理机上程序运行的软件环境是怎么样？

2021-09-06 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 18:55 大小 17.34M



你好，我是宫文学。

上一节课，我们主要讨论了程序运行的硬件环境。在某些编程场景下（比如嵌入式编程），我们的语言需要直接跑在裸设备上。所以，你要能够理解在裸设备上运行某个语言的程序所需要的技术。

不过，现代语言大部分情况下都运行在某个操作系统里，操作系统为语言提供了基础的运行环境，比如定义了可执行文件的格式、程序在内存中的布局、内存管理机制，还有并发机制等等。计算机语言需要跟操作系统紧密配合，才能更好地运行。



今天这节课，我们就来讨论一下操作系统中与计算机语言有关的那些知识点，包括内存管理、任务管理和 ABI，为进一步实现我们的计算机语言打下良好的基础。

首先我们来看看操作系统的内存管理功能和程序的关系。

内存管理

操作系统的一个重要功能就是管理内存，它会把内存虚拟化，并进行内存访问的权限管理。

那什么是虚拟化呢？

虚拟化就是让每个进程都有一个自己可以用的寻址空间，不过这些地址是假的地址。但就这些假地址，我们通过指令发给 CPU，CPU 也是认的。因为 CPU 中有一个内存管理单元，缩写是 MMU，它能够根据这个逻辑地址，计算出在内存里真实的物理地址。MMU 还可以跟操作系统配合，设置每个内存页面的权限，包括是否可读、可写和可执行。

这种逻辑地址与物理地址转换的功能，对我们做编译很有用。我们的程序编译成目标代码的时候，里面的每个函数、常量和全局变量的地址，都是确定的。这样，在操作系统加载了可执行程序以后，就可以正确地调用每个函数，访问每个数据了。

如果没有这个逻辑地址的功能，那运行多个程序就困难了。我们要保证每个程序使用的地址都互相不冲突才行，否则大家就乱了套了。在不使用操作系统的编程领域（比如某些嵌入式编程），重点就要解决好这些问题。

在虚拟化机制下，只有你用到某个地址，操作系统才会为这个地址分配真实的物理内存。这些物理内存一般划分成页来管理，MMU 会根据这些分页信息把逻辑地址转换成物理地址。

这里我们横向延展一下，我们在 [第 11 节课](#) 说过，在栈里申请内存的时候很简单，只需要移动一下栈顶指针就行了，其实这个内部机制和我们上面说的虚拟化也是有关的。

在 X64 架构下，栈顶指针使用的是 `rsp` 寄存器。但其实，这时候并没有真正分配内存，你只是改变了寄存器的值而已。但如果你访问栈里的某个地址，而且这个地址又没有分配物理的内存页，那么 CPU 在访问内存的时候就会知道这里出错了。它就会触发一个缺页中断，跳转到中断处理程序，去分配页面。分配完毕以后，又跳回原来的程序接着执行，我们的程序并不知道背后发生了这么多的事情。

说到中断，我再额外跟你补充一点。CPU 在处理中断的时候，要保护当前的现场，比如各个寄存器的状态。否则，如果各个寄存器的值被弄乱了，原来的程序就没法执行了。然后在返回原来的程序的时候，CPU 要恢复现场。整个过程对我们的程序是透明的。

所以说，这个 `rsp` 只是起到一个标记作用，是我们的程序跟操作系统之间的一个约定。我们只要修改了 `rsp` 里的值，操作系统就要保证给我们提供足够的内存。如果你违背了这个约定去乱访问一些地址，在 MMU 和操作系统的配合下，也会被识别出来，就会报内存访问的错误。

既然这只是个约定，那么有的操作系统就比较为程序着想了，它规定你可以访问栈顶之外的一定范围内的内存。比如，Linux 和大多数类 Unix 的系统都遵循 System V AMD64 ABI，它规定可以访问栈顶之外的 128 个字节范围内的内存。

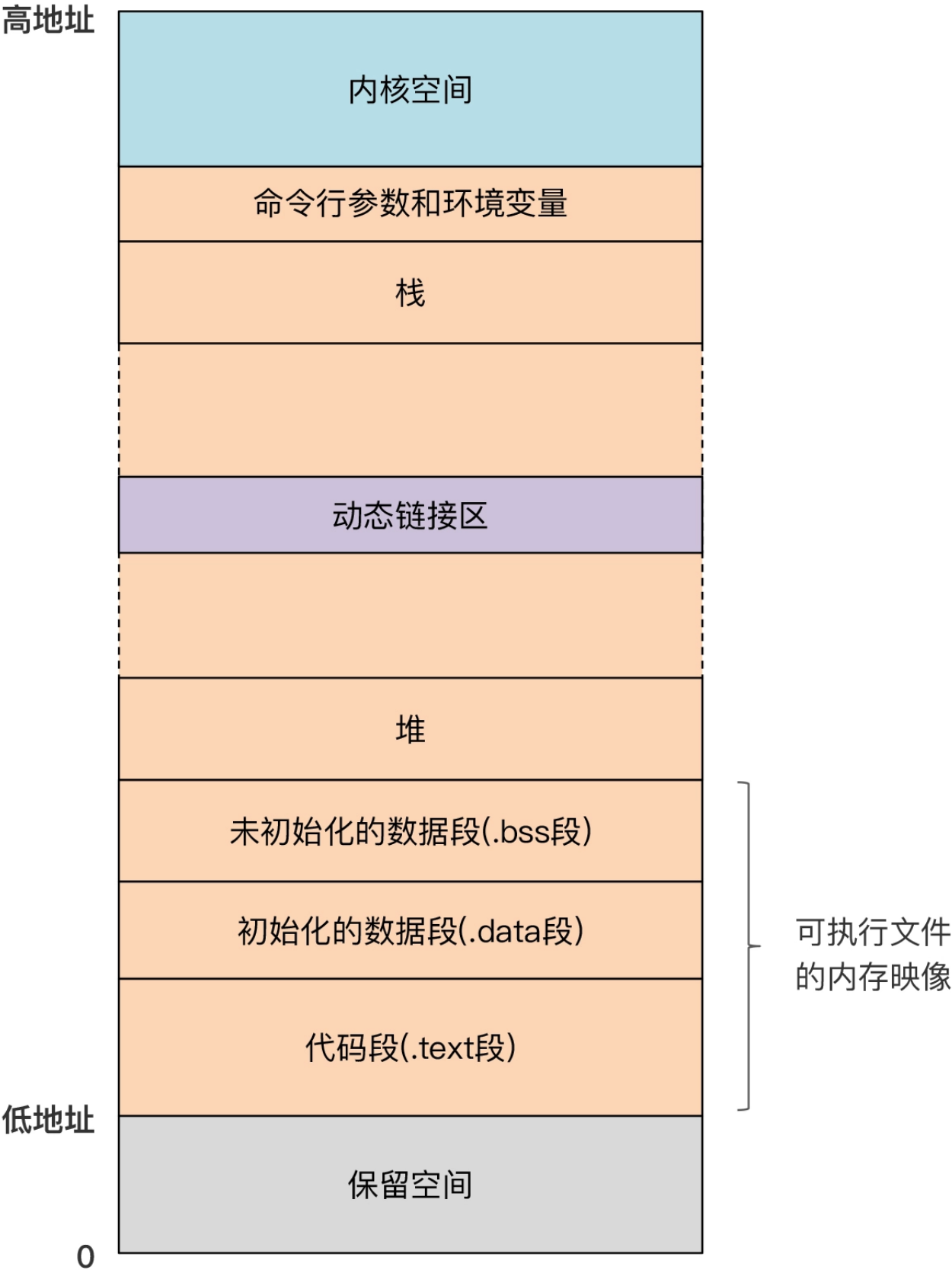
这有什么好处呢？好处是，对于程序中的叶子函数，也就是这个函数没有调用其他函数，并且它所使用的数据不会超过 128 个字节的情况，我们根本不需要去建立栈帧，也就省去了把栈顶指针的值保存到内存，修改栈顶指针，最后再从内存中恢复栈顶指针这一系列操作，这样就节省了大量的内存读写时间，让系统性能得到优化。

在堆中申请内存也是一样，也不是真实的分配，只是提供了一些标记信息，之后可供 MMU 使用而已。

好了，关于从栈和堆里申请内存的延伸就到这里，我们回归主线，继续来看虚拟化机制带来的结果。简单的说，虚拟化机制，可以让运行中的各种程序，使用相同的逻辑地址，但实际上对应的是不同的物理地址。这样各个程序加载到内存后，就都可以使用标准的内存布局。

那一个可执行程序在内存中的布局情况是怎样的呢？我们用一个 C 语言的程序的例子来分析一下。

在 Linux 或 macOS 系统中，一个 C 语言的程序加载到内存以后，它的内存布局大概是下面的样子：



你可以看到，其中代码段（.text）和数据段（包括.data 和.bss）是从可执行文件直接加载进内存的。可执行文件中提前计算好的函数、常量和全局变量的地址，也就变成了内存中


的地址。

另外两个重要的区域是栈和堆。栈是从高地址向低地址延伸的，而堆则是从低地址向高地址延伸的。

但是，在不同的操作系统中，上图中每个部分的具体地址都是不大相同的，比如，macOS 和 Linux 的就不同。不过，你可以写个程序，打印出不同区域中的地址。我写了个示例程序 address.c，你可以参考一下，这里你要好好琢磨一下示例代码中每个地址是如何获取的。

另外，我是在 macOS 上运行的，如果你用的是不同的操作系统，也可以运行一下，看看打印出来的地址跟我的有什么区别。

这个程序是这样的：

 复制代码

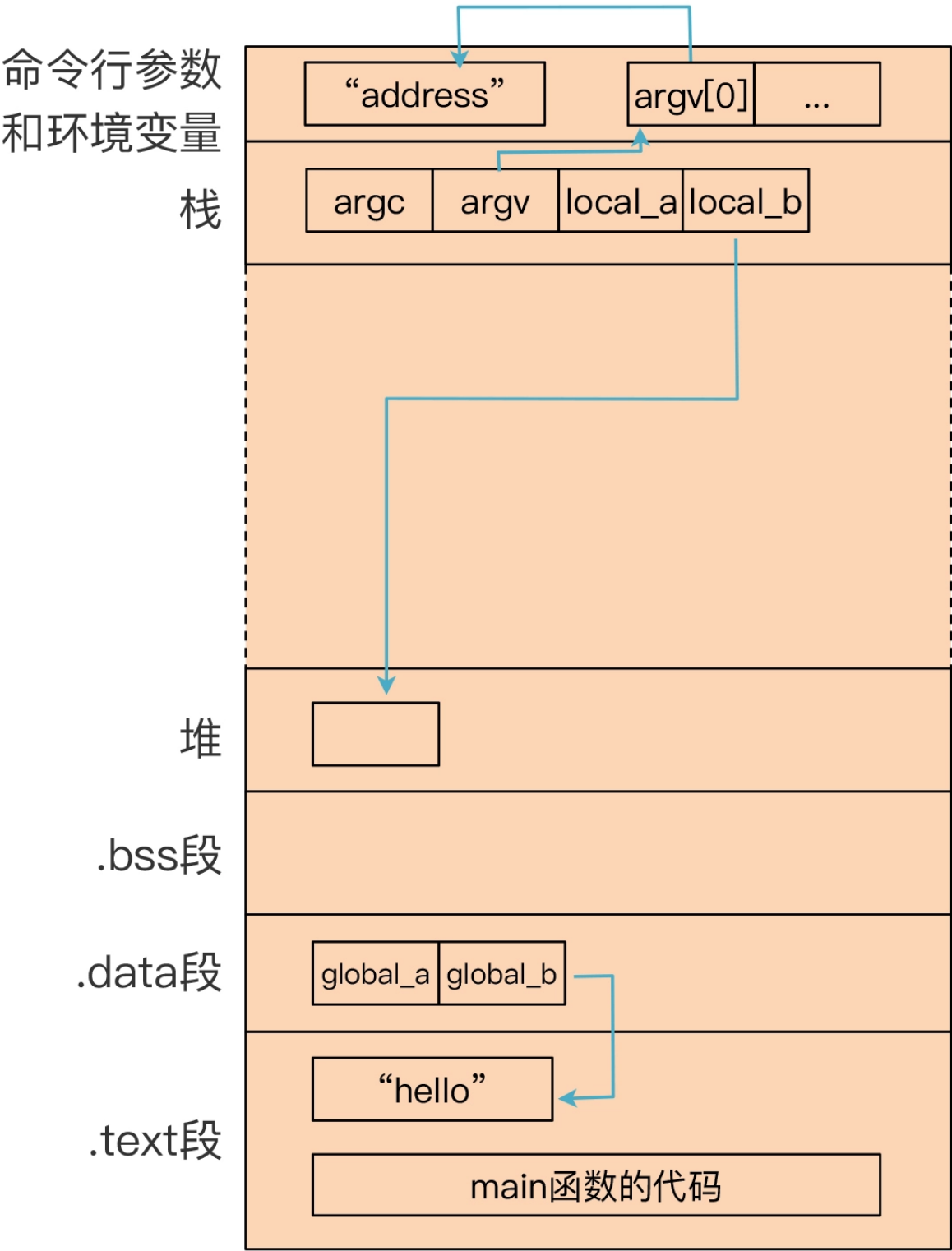
```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 //全局变量
5 int global_a = 10;
6 char * global_b = "hello";
7
8 int main(int argc, char** argv){
9     printf("命令行参数/\"address\"的地址:\t0x%12lX\n", (size_t)argv[0]);
10    printf("命令行参数/argv数组的地址: \t0x%12lX\n", (size_t)argv);
11
12    printf("栈/参数argc的地址: \t0x%12lX\n", (size_t)&argc);
13    printf("栈/参数argv的地址: \t0x%12lX\n", (size_t)&argv);
14
15    int local_a = 20;
16    printf("栈/local_a的地址: \t0x%12lX\n", (size_t)&local_a);
17
18    int * local_b = (int*)malloc(sizeof(int));
19    printf("栈/local_b的地址: \t0x%12lX\n", (size_t)&local_b);
20    printf("堆/local_b指向的地址: \t0x%12lX\n", (size_t)local_b);
21    free(local_b);
22
23    printf("data段/global_b的地址: \t0x%12lX\n", (size_t)&global_b);
24
25    printf("data段/global_a的地址: \t0x%12lX\n", (size_t)&global_a);
26
27    printf("text段/\"hello\"的地址: \t0x%12lX\n", (size_t)global_b);
28}
```

```
29     printf("text段/main函数的地址:    \t0x%12lX\n", (size_t)main);
30 }
31
```

示例程序的运行结果如下，我是按照高地址到低地址的顺序打印各个不同的地址的：

```
→ 12 git:(master) × clang address.c -o address
→ 12 git:(master) × ./address
命令行参数/"address"的地址:    0x7FFEE3D7D968
命令行参数/argv数组的地址:    0x7FFEE3D7D7F0
栈/参数argc的地址:    0x7FFEE3D7D7CC
栈/参数argv的地址:    0x7FFEE3D7D7C0
栈/local_a的地址:    0x7FFEE3D7D7BC
栈/local_b的地址:    0x7FFEE3D7D7B0
堆/local_b指向的地址:    0x7FD9B9D059C0
data段/global_b的地址:    0x    10BE8A028
data段/global_a的地址:    0x    10BE8A020
text段/"hello"的地址:    0x    10BE85DE2
text段/main函数的地址:    0x    10BE85C70
```

我也画了一个示意图：



你看这张图，最上面的是命令行参数。main 函数的参数 argv[0]，实际上是操作系统提供的一个命令行参数，也就是可执行文件的名称，而 argv[0]的值，就是字符串 “address” 的地址。

顺着看下来，下面的是两个参数。参数是被放在栈里的，第一个参数排在前面，第二个参数排在后面。其中第二个参数是 `argv`，它的值是指向一个参数数组。

接下来是两个本地变量 `local_a` 和 `local_b`，它们也是放在栈里的。其中 `local_b` 是一个指针，指向堆里申请的一小块内存的地址。

再往下是两个全局变量，它们存在 `.data` 段。其中 `global_b` 是一个指针，指向一个字符串常量 `"hello"`。

最后是 `.text` 段，这里有字符串常量 `"hello"`，也有 `main` 函数对应的机器码。

你看，这样小的程序，都涉及了这么多个地址。全局变量、常数和代码的地址是由编译和链接程序确定的，命令行参数的地址是由操作系统分配的，而函数参数、本地变量和从堆中申请的内存，是在运行期分配的。

那接下来，一个问题就来了：无论哪种语言，它的内存布局都是这样的吗？

也不完全是。像 C、C++ 这些静态编译的语言，要提前编译成可执行文件，把可执行文件加载到内存以后，内存布局就是上面的样子。其中，可执行文件的格式、如何加载到内存、每个段的起始地址是多少、如何让程序能够访问环境变量和命令行参数、调用系统 API 时的调用约定等等，都是由操作系统的 ABI 规定的。

但也有的语言，特别是带有即时编译能力的语言，以及想要实现自己的并发机制的语言，会用更灵活的方式来使用内存。

我拿即时编译的情况举个例子。现代很多语言是编译成机器码运行的，但却不是提前编译的，而是在运行时编译的，典型的就 Java 和基于 V8 的 JavaScript。既然是即时编译成机器码的，那也就不可能像 C 语言那样提前计算好每个函数的代码地址，并且放到内存中的文本段，而是动态地从堆中申请内存，来保存这些机器码，并且运行它们。

这里我用一个示例程序来给你演示一下即时编译的原理。这个程序能够把一段二进制的机器码加载到内存，然后运行它。

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


[复制代码](#)


```
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/mman.h>
6
7 /*
8  * 机器码，对应下面函数的功能：
9  * int foo(int a){
10  *     return a + 2;
11  * }
12  */
13 uint8_t machine_code[] = {
14     0x55, 0x48, 0x89, 0xe5,
15     0x8d, 0x47, 0x02, 0x5d, 0xc3
16 };
17
18 //加载并运行机器码。
19 int main() {
20     //分配一块内存，设置权限为读和写
21     void *mem = mmap(NULL, sizeof(machine_code), PROT_READ | PROT_WRITE,
22                      MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
23     if (mem == MAP_FAILED) {
24         perror("mmap");
25         return 1;
26     }
27
28     //把机器码写到刚才的内存中
29     memcpy(mem, machine_code, sizeof(machine_code));
30
31     //把这块内存的权限改为读和执行
32     if (mprotect(mem, sizeof(machine_code), PROT_READ | PROT_EXEC) == -1) {
33         perror("mprotect");
34         return 2;
35     }
36
37     //用一个函数指针指向这块内存，并执行它
38     int32_t(*fn)(int32_t) = (int32_t(*) (int32_t)) mem;
39     int32_t result = fn(1);
40     printf("result = %d\n", result);
41
42     //释放这块内存
43     if (munmap(mem, sizeof(machine_code)) == -1) {
44         perror("munmap");
45         return 3;
46     }
47
48     return 0;
49 }
50
```

整个程序的逻辑很简单。我们声明了一个数组，里面保存了几个机器码指令，然后申请了一块内存，把这些机器码拷贝到这片内存，并给这片内存设置了可读和可执行的权限。

接着，神奇的事情发生了。我们把这块内存的指针，强制转化成了一个函数指针，那么我们就可以把这段代码当做一个函数来执行，并正确地返回执行结果了！

示例程序中的机器码指令相当于下面这个函数编译后的结果：

 复制代码

```
1 int foo(int a){  
2     return a + 2;  
3 }
```

那这个例子说明了什么呢？

一方面，对于计算机来说，代码跟数据没啥区别。对于编译器来说，这些机器码是编译器所生成的数据。而对于 CPU 来说，这是它可以认识并执行的指令，你只要喂给它，它就能执行这行指令。

另一方面，作为语言的实现者，你可以从这个例子中发现，其实你拥有很大的自由去决定如何使用内存，在哪儿放数据，在哪儿放代码，从而可以采取最佳的技术策略来实现你的设计目标。

好，关于内存管理我们就先说这么多。现在我们来看操作系统的另一个重要功能：任务管理。

任务管理

任务管理，我们主要指的就是多任务的处理，现代计算机都支持多任务的处理。在上一节课，我们讲了 CPU 为多任务处理所提供的硬件支撑，通过多处理器、多核以及超线程技术，我们可以让多个任务并行地执行。在硬件能力较低的情况下，哪怕我们只有一条真实的执行线索，我们仍然可以借助硬件的中断机制（主要是时钟中断）让多个任务分时执行，从而实现并发机制。

所以说，以这些硬件的功能为基础，其实即使没有操作系统的支持，作为语言的编写者，其实你也有办法让程序支持并行和并发。

但在操作系统的支持下，让程序实现多任务处理会更简单一些。

因为操作系统在硬件提供的能力的基础上，封装出了进程、线程这样的并发调度机制，让我们可以忽略很多任务管理中的细节。比如，进程和线程都对应着一些上下文信息，像寄存器之类的信息，等等。在进行任务的切换时，操作系统会为我们的程序自动保存和恢复这些上下文信息，调度过程对我们的程序来说是透明的，编译器不需要做什么额外的工作。

但是，云计算时代的到来，使得并发任务越来越多，仅仅靠操作系统级提供的并发机制已经不能满足我们的要求了。所以，在语言层面，各个语言也开始提供并发机制，比如协程机制、Actor 机制等。

在语言层面上实现的并发机制，和操作系统级别的机制，是有所区别的。

操作系统的多任务机制，一般是抢占式的。也就是说，操作系统可以强行把当前任务调度走，去执行其他任务，像 Linux 和 Windows 系统，都是抢占式的。这个过程需要用到中断机制，由操作系统内核态的代码去处理中断。

而用户的程序是运行在用户态的，所以由计算机语言本身支持的并发机制，一般是协作式的，因为我们没有办法打断别的任务的运行。所以，如果一个协程一直不停下来，不主动交出 CPU 来，我们通常是没有办法干涉的。

但是，编译器和运行时配合，我们仍然是可以做一些工作，避免某些任务长时间占据 CPU。比如，Erlang 是基于虚拟机运行的，这时候虚拟机就像一个物理的 CPU 一样，检查某个并发任务执行了多长时间，需要时把它调度走。

而 Go 语言的 Goroutine，是编译成本地代码运行的，并没有一个解释器来控制代码的执行，但它也会在编译后的程序里插入一些代码，用于并发调度。程序在执行到这里的时候，运行时会去检查 Goroutine 是否超时了，超时就把它挂起来，或者把一些等候过久的任务挪到比较空闲的线程去执行。

并且，在语言级实现的多任务机制，也涉及任务的数据和状态信息的管理问题。

比如，操作系统中，只为每个线程准备了一个栈，随着函数的逐级调用，栈会不停地伸缩。而采用了协程以后，有多个并发的任务在执行，每个任务都会形成自己的调用栈。这个时候，语言的运行时就要打破操作系统传统的栈管理机制，形成自己的管理机制。

其实这个方法我们也不陌生，在前一节课实现基于 C 语言的虚拟机的时候，栈帧不就是我们自己管理的吗？这也再次印证了前面我们内存管理部分提到的内容，也就是计算机语言的设计者有很大的自由度决定如何使用内存。

总结起来，现代计算机支持多任务，通常需要 CPU、操作系统和计算机语言（含编译器和运行时）三方的通力合作才行。当谈论到并发的时候，你要知道哪些是计算机语言的作者可以控制的，尽量发挥我们的创造性，可以在不少领域产生很好的成果。

好了，我们已经讨论了操作系统最重要的两个作用，以及它们和我们实现计算机语言的关系。但除了这两点大的内容以外，还有其他一些内容，它们其实可以被归结在 ABI 里面。

ABI

ABI，我们前面讲过，是 Application Binary Interface 的意思，也就是应用程序的二进制接口。对谁的接口呢？对操作系统的。所以，操作系统都提供了二进制接口方面的规范。

在上一节课，我建议大家多看 CPU 的手册，来理解程序运行的硬件环境。而要全面了解程序的软件环境，其实也是查手册就行了，这里最重要的就是 ABI 的手册。像 Linux 和 macOS 这些类 Unix 的操作系统，遵循的都是 SYSTEM V APPLICATION BINARY INTERFACE。为了方便你的阅读，我把一些常用的 ABI 文档上传到了码云，你一定要打开看看。

ABI 文档里包含的内容很多，基本上，要让一个二进制程序能够在某个操作系统上跑起来所需要的所有信息，都在这里规定了。比如：

对二进制文件格式的规定；

如何加载程序，以及实现动态链接；

在系统里可以用到的标准库；

等等。

还有一些内容，要根据所采用的 CPU 架构来做规定。比如，在针对 AMD64 架构的补充手册（System V Application Binary Interface AMD64 Architecture Processor Supplement）中，有一个 3.2 节，标题是函数调用顺序（Function Calling Sequence），就规定了栈帧的结构、如何传递参数、哪些寄存器是由被调用者负责保护的等等，这些信息在我们后面生成汇编代码的时候特别有用。

这些规定，有些是我们必须遵守的，比如在调用函数时对栈帧的内存对齐的规定，否则操作系统就会报错。有些呢，我们语言的设计者可以不用严格遵守，为我们的语言发明我们自己的机制，就像函数如何传参、如何接收返回值等调用约定。比如，Go 语言支持多个返回值，显然就跟 C 语言的机制是不同的。但是，如果我们要调用系统的标准库，那么就必须遵守 ABI 里所规定的调用约定才行。

课程小结

今天这节课，我们总结了程序运行与操作系统的关系，了解了在设计自己的语言时，如何充分利用操作系统所提供的能力，而且充分运用作为语言的作者可发挥的空间。

操作系统最大的作用，是为我们的程序提供了**内存管理**和**任务管理**的支持。

从内存管理方面，操作系统为应用程序提供了一个逻辑的地址空间，使得运行中的各种程序，虽然使用相同的逻辑地址，但实际的物理地址都是不同的。这样各个程序加载到内存后，就都可以使用标准的内存布局。不过，在采用 JIT 机制时，语言的设计者也可以设计自己的内存布局。

在任务管理方面，操作系统在底层硬件提供的功能之上，封装成了进程和线程的并发调度机制，计算机语言可以和这些并发机制相配合，并提供应用级的、协作式的、更加轻量级的并发能力。

最后，**如果要充分吃透程序运行的操作系统环境，你一定要阅读 ABI 手册**。你在教科书上找不到的知识点，往往能在手册里找到准确的答案。

思考题

请你思考一下，把一个程序编译成在没有操作系统的计算机上运行，会跟有操作系统的计算机上运行有啥不同？你可以仔细思考一下各方面的因素，这会有利于你加深对程序运行机制的理解。

感谢你和我一起学习，也欢迎你把这节课分享给更多对物理计算机的运行机制感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. [🔗 SYSTEM V APPLICATION BINARY INTERFACE Edition 4.1](#) 这是 Linux、macOS 等类 Unix 的系统都遵守的二进制接口规范。
2. [🔗 SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement](#)，这是上面 ABI 针对 Intel 架构 CPU 的补充规定。
3. [🔗 System V Application Binary Interface AMD64 Architecture Processor Supplement](#)，这是上面 ABI 针对 X86 的 64 位模式（叫做 AMD64，因为 AMD 公司最先做出了 64 位模式 X86 芯片）的补充规定。
4. [🔗 Executable and Linking Format \(ELF\) specification v1.2](#)，这是 Linux 二进制目标文件的格式标准。
5. [🔗 Formats Specification for Windows v1.0](#)，这是 Windows 二进制目标文件的格式标准。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

👍 赞 0 📝 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 12 | 物理机上程序运行的硬件环境是怎么样？

精选留言 (1)

写留言



罗乾林

2021-09-06

没有操作系统环境下：

1、程序编译时需要指定程序链接时的地址布局

2、运行时需要将程序加载到对应的物理地址再从指定地址运行

...

👍