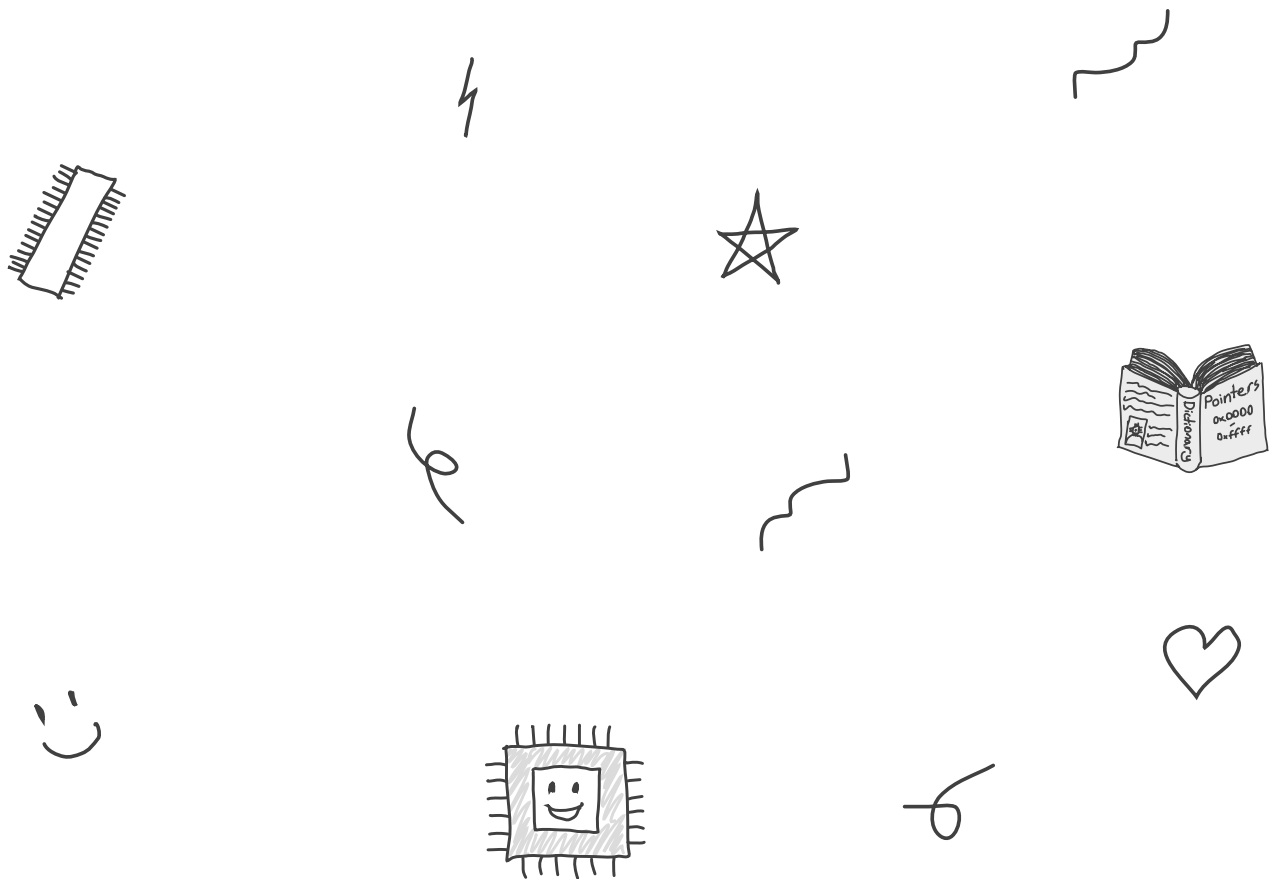




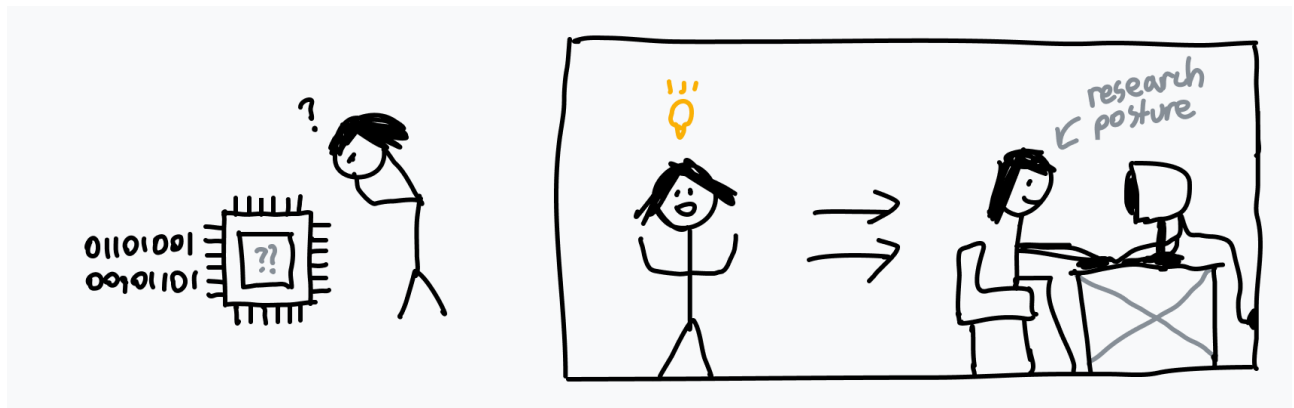
# Putting the “You” in CPU

By Lexi Mattick & Hack Club · July, 2023



# Chapter 0: Intro

我在计算机领域做过许多事情,但我的知识中一直存在一个空白:当你在电脑上运行一个程序时,究竟发生了什么?我思考这个问题时发现 - 我掌握了大部分必要的底层知识,但难以将所有碎片拼凑起来。程序真的是直接在CPU上执行的吗,还是另有玄机?我使用过系统调用,但它们究竟是如何工作的?它们本质上是是什么?多个程序是如何同时运行的?



我终于忍不住开始尽可能地弄清楚这些问题。如果你不上大学,就很难找到全面的系统资源,所以我不得不翻阅大量质量参差不齐、有时甚至相互矛盾的资料。经过几周的研究和近40页笔记后,我认为我对计算机从启动到程序执行的工作原理有了更深入的理解。如果当时有一篇扼要解释我所学内容的文章就好了,所以我决定写下这篇我曾渴望拥有的文章。

你知道人们常说的那句话吗...”只有当你向别人解释清楚时,你才是真正理解了某样东西。”

赶时间?觉得自己已经了解这些内容了?

**[阅读第三章]**,我保证你会学到新东西。除非你是 Linus Torvalds 本人。

# Chapter 1: The “Basics”

在写这篇文章的过程中,一件反复让我惊讶的事情是计算机实际上有多简单。我仍然很难不过度思考,期望比实际存在的更多的复杂性或抽象!如果在继续之前你应该牢记一件事,那就是看似简单的东西实际上就是那么简单。这种简单性非常美丽,有时也非常非常诅咒。

让我们从计算机最核心的工作原理开始。

## 计算机的架构方式

计算机的中央处理器(CPU)是计算的大脑。它是核心中的核心,是整个系统的灵魂所在。一旦你启动计算机,它就开始不知疲倦地工作,一条接一条地执行指令。

第一个大规模生产的CPU是[Intel 4004](#),由意大利物理学家兼工程师Federico Faggin在60年代末设计。它是4位架构,不同于我们今天使用的[64位](#)系统,功能也远不如现代处理器复杂。但是,它的许多基本原理至今仍在使用。

CPU执行的“指令”其实就是二进制数据:一两个字节表示要执行的指令(操作码),后面跟着执行指令所需的数据。我们所说的机器码不过是一系列连续的二进制指令。[汇编语言](#)是一种更易于人类阅读和编写的语法,用于表示机器码;它最终总是被编译成CPU能够理解的二进制代码。



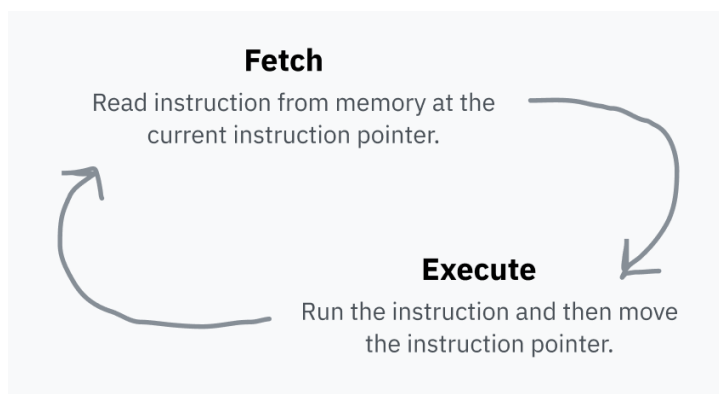
补充说明：指令在机器码中并不总是一一对应的。比如，`add eax, 512`会被翻译成`05 00 02 00 00`。

第一个字节（`05`）是一个特定的操作码，表示将一个32位数加到EAX寄存器上。剩下的字节是512（`0x200`），按小端序排列。

Defuse Security开发了一个很有用的工具，可以让你玩玩汇编和机器码之间的转换。

内存（RAM）是计算机的主要存储区，是一个大型多用途空间，存储了计算机上运行的程序使用的所有数据。这包括程序代码本身，以及操作系统核心的代码。CPU总是直接从内存中读取机器码，如果代码没有加载到内存中，就无法运行。

CPU有一个指令指针，指向内存中下一条要执行的指令的位置。执行每条指令后，CPU移动指针并重复这个过程。这就是我们所说的取指-执行循环。



执行指令后，指针向前移动到内存中紧接在指令之后的位置，指向下一条指令。这就是代码运行的原理！指令指针不断向前推进，按照存储在内存中的顺序执行机器码。某些指令可以告诉指令指针跳到其他地方，或者根据特定条件跳到不同的地方；这使得代码复用和条件逻辑成为可能。

这个指令指针存储在一个叫做寄存器的地方。寄存器就像是CPU的小本子，读写速度极快。每种CPU架构都有固定的寄存器集，用途从存储计算过程中的临时值到配置处理器等各种任务。

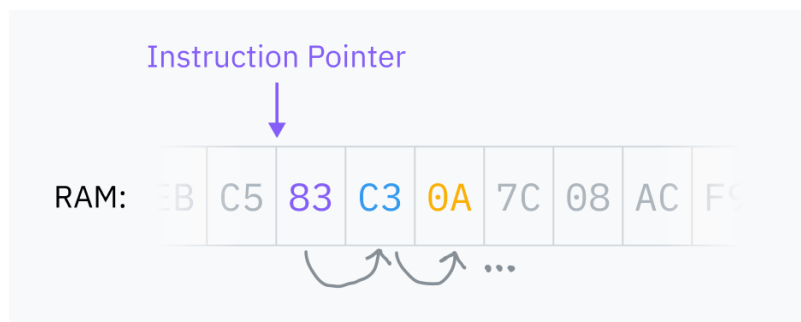
有些寄存器可以直接通过机器码访问，比如前面图表中的`ebx`。

其他寄存器只在CPU内部使用，但通常可以通过特殊指令更新或读取。比如指令指针，虽然不能直接读取，但可以通过跳转指令等方式更新。

## 处理器其实很单纯

让我们回到最初的问题：当你在电脑上运行一个程序时，究竟发生了什么？首先，有一系列复杂的准备工作——我们稍后会详细讲解——但最终，某个文件中会有机器码。操作系统将这些代码加载到内存中，然后告诉CPU把指令指针指向内存中的那个位置。CPU继续它的取指-执行循环，于是程序就开始运行了！

（这是那种让我惊讶的时刻之一——说真的，你现在用来阅读这篇文章的程序就是这样运行的！你的CPU正在从内存中依次获取你的浏览器的指令并直接执行它们，它们正在渲染这篇文章。）



事实证明，CPU的世界观非常简单；它只关注当前的指令指针和一些内部状态。进程这个概念完全是操作系统的抽象，不是CPU本身理解或跟踪的东西。

*\*打个比方\* 进程就像是操作系统编造出来的童话故事，为的是卖更多的电脑*

对我来说，这引发了比它回答的更多的问题：

1. 如果CPU不知道多进程这回事，只是一条接一条地执行指令，为什么它不会被困在某个程序里出不来？多个程序是如何同时运行的？
2. 如果程序直接在CPU上运行，而CPU可以直接访问内存，为什么代码不能访问其他进程的内存，或者更可怕的是，访问内核的内存？
3. 说到这里，有什么机制可以防止每个进程随意执行指令，对你的计算机为所欲为？还有，系统调用到底是个什么东西？

关于内存的问题值得单独讨论，我们会在[第5章]详细解释——简单来说，大多数内存访问实际上都经过了一层转换，重新映射了整个地址空间。现在，我们先假设程序可以直接访问所有内存，计算机一次只能运行一个进程。我们稍后会解释这两个假设。

现在，让我们深入探索系统调用和安全环这个复杂而有趣的领域。

## 顺便说一下，什么是内核？

你电脑上的操作系统，无论是macOS、Windows还是Linux，都是一组软件的集合，负责让你的电脑正常工作。“正常工作”这个词很笼统，“操作系统”这个词也是——取决于你问谁，它可能包括电脑自带的应用程序、字体和图标等。

但是，内核是操作系统的核心。当你开机时，指令指针会指向某个程序的起点。那个程序就是内核。内核几乎可以完全控制你电脑的内存、外设和其他资源，负责运行安装在你电脑上的软件（我们称之为用户态程序）。在接下来的内容中，我们会了解内核是如何获得这种控制权的——以及为什么用户态程序没有这种权限。

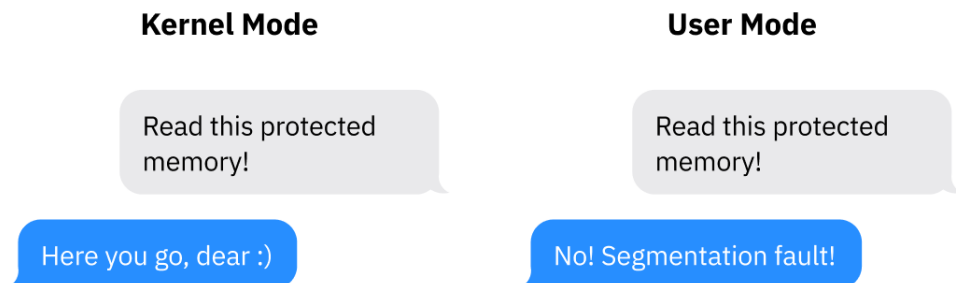
Linux本身只是一个内核，需要配合大量用户空间的软件如shell和显示服务器才能使用。macOS中的内核叫做XNU，是类Unix的，而现代Windows的内核叫做NT Kernel。

## 两个世界：内核模式和用户模式

处理器的模式（有时也叫特权级别或环）决定了它能做什么。现代架构至少有两种模式：内核/监管模式和用户模式。虽然一个架构可能支持更多的模式，但现在通常只用内核模式和用户模式。

在内核模式下，处理器可以为所欲为：可以执行任何支持的指令，访问任何内存。而在用户模式下，只能执行部分指令，输入输出和内存访问都受限，许多CPU设置也被锁定。一般来说，内核和驱动程序在内核模式下运行，而应用程序在用户模式下运行。

处理器启动时是在内核模式。在执行程序之前，内核会切换到用户模式。



在实际的CPU架构中，这是怎么实现的呢？以x86-64为例，当前特权级别（CPL）存储在一个叫做cs（代码段）的寄存器中。具体来说，CPL就是cs寄存器的两个最低有效位。这两个位可以表示x86-64

的四个可能的环：环0是内核模式，环3是用户模式。环1和环2本来是为驱动程序设计的，但现在只有少数老旧的小众操作系统还在使用。比如，如果CPL位是11，那么CPU就是在环3（也就是用户模式）下运行。

## 系统调用是个什么东西？

程序在用户模式下运行，是因为我们不能完全信任它们。用户模式阻止了程序访问大部分计算机资源——但程序总需要进行输入输出、分配内存，以及与操作系统交互吧！为此，用户模式下运行的软件必须向操作系统内核求助。然后操作系统可以实施自己的安全措施，防止程序做坏事。

如果你写过与操作系统交互的代码，你可能认识open、read、fork和exit这样的函数。在几层抽象之下，这些函数都使用系统调用来向操作系统求助。系统调用是一种特殊的程序，允许程序从用户空间跳到内核空间，从程序的代码跳到操作系统的代码。

从用户空间到内核空间的控制转移是通过处理器的一个叫做[软件中断](#)的功能实现的：

1. 在开机过程中，操作系统在内存中创建一个叫做[中断向量表](#)的东西（在x86-64中叫做[中断描述符表](#)），并告诉CPU。这个表把中断号和处理程序的代码地址对应起来。

Interrupt Vector Table	
#	Handler Address
01	0x3A28213A6339392C
02	0x7363682EEE208A47
03	0x2B290904B9B89815
04	0xF97CA091A8D9B16C
So on and such forth...	

2. 然后，用户程序可以使用像[INT](#)这样的指令，告诉处理器在中断向量表中查找给定的中断号，切换到内核模式，然后把指令指针跳到表中存储的内存地址。

当内核代码完成后，它会使用像[IRET](#)这样的指令告诉CPU切换回用户模式，并把指令指针返回到触发中断时的位置。

（如果你好奇，Linux上用于系统调用的中断ID是0x80。你可以在[Michael Kerrisk的在线手册页目录](#)上查看Linux系统调用列表。）



# Chapter 2: Slice Dat Time

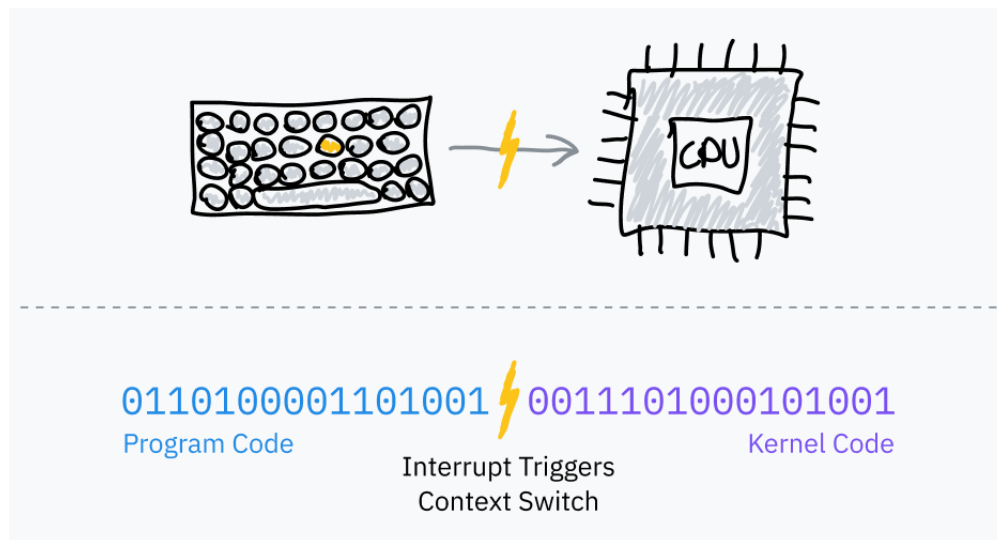
假设你正在构建一个操作系统，你希望用户能够同时运行多个程序。但是你没有fancy的多核处理器，所以你的CPU一次只能执行一条指令！

幸运的是，你是一个非常聪明的操作系统开发者。你想出了一个办法，可以通过让进程轮流使用CPU来模拟并行性。如果你循环遍历这些进程，每次执行其中几条指令，它们就都能保持响应，而不会有单个进程霸占CPU。

但是你如何从程序代码中夺回控制权来切换进程呢？经过一番研究，你发现大多数计算机都配备了定时器芯片。你可以编程定时器芯片，在一定时间后触发切换到操作系统中断处理程序。

## 硬件中断

早些时候，我们讨论了如何使用软件中断将控制权从用户程序交给操作系统。之所以称为“软件”中断，是因为它们是由程序自愿触发的——处理器在正常的取指-执行循环中执行的机器码告诉它将控制权切换到内核。



操作系统调度器使用像[PITs](#)这样的定时器芯片来触发硬件中断以实现多任务：

1. 在跳转到程序代码之前，操作系统设置定时器芯片在一定时间后触发中断。
2. 操作系统切换到用户模式并跳转到程序的下一条指令。
3. 当定时器时间到了，它触发一个硬件中断，切换到内核模式并跳转到操作系统代码。

4. 此时操作系统可以保存程序离开时的状态，加载另一个程序，并重复这个过程。

这被称为*抢占式多任务*；对进程的中断被称为*抢占*。如果你正在浏览器上阅读这篇文章，同时在一台机器上听音乐，你的电脑很可能每秒都在成千上万次地重复这个循环。

## 时间片计算

*时间片*是操作系统调度器允许一个进程运行的时间，之后会抢占它。选择时间片最简单的方法是给每个进程相同的时间片，可能在10毫秒左右，然后按顺序循环执行任务。这被称为*固定时间片轮询调度*。

### 题外话：有趣的专业术语！

你知道时间片经常被称为“量子”吗？现在你知道了，你可以用这个知识来打动你所有的技术朋友。我觉得我值得被大加赞赏，因为我没有在这篇文章的每隔一句话就提到量子。

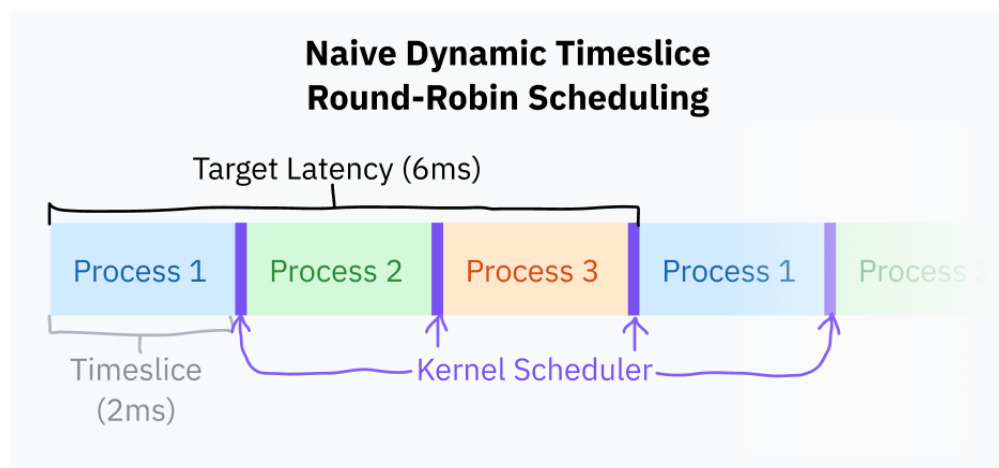
说到时间片的专业术语，Linux内核开发者使用*jiffy*时间单位来计算固定频率的定时器滴答。除此之外，*jiffies*还用于测量时间片的长度。Linux的*jiffy*频率通常是1000 Hz，但可以在编译内核时配置。

固定时间片调度的一个小改进是选择一个*目标延迟*——进程响应的理想最长时间。目标延迟是进程被抢占后恢复执行所需的时间，假设有合理数量的进程。*这很难想象！别担心，马上会有一个图表。*

时间片的计算方法是将目标延迟除以任务总数；这比固定时间片调度更好，因为它在进程较少时消除了浪费的任务切换。如果目标延迟为15毫秒，有10个进程，每个进程将得到15/10或1.5毫秒的运行时间。如果只有3个进程，每个进程会得到更长的5毫秒时间片，同时仍然达到目标延迟。

进程切换在计算上是昂贵的，因为它需要保存当前程序的整个状态并恢复另一个程序的状态。过了某个点，太小的时间片可能会导致进程切换过于频繁的性能问题。通常会给时间片持续时间设置一个下限（*最小粒度*）。这确实意味着当有足够多的进程使最小粒度生效时，会超过目标延迟。

在撰写本文时，Linux的调度器使用6毫秒的目标延迟和0.75毫秒的最小粒度。



使用这种基本时间片计算的轮询调度接近现在大多数计算机的做法。但它仍然有点naive；大多数操作系统倾向于使用更复杂的调度器，考虑进程优先级和截止时间。自2007年以来，Linux一直使用一个称为[完全公平调度器](#)的调度器。CFS做了一堆非常fancy的计算机科学的事情来优先处理任务并分配CPU时间。

每次操作系统抢占一个进程时，它都需要加载新程序保存的执行上下文，包括其内存环境。这是通过告诉CPU使用不同的页表来完成的，页表是从“虚拟”地址到物理地址的映射。这也是防止程序访问彼此内存的系统；我们将在本文的[\[第5章\]](#)和[\[第6章\]](#)中深入探讨这个问题。

## 注1：内核可抢占性

到目前为止，我们只讨论了用户程序的抢占和调度。如果内核代码处理系统调用或执行驱动程序代码花费太长时间，可能会让程序感觉卡顿。

现代内核，包括Linux，都是[可抢占内核](#)。这意味着它们以一种允许内核代码本身被中断和调度的方式编程，就像用户程序一样。

除非你正在编写内核或类似的东西，否则了解这一点并不是很重要，但基本上我读过的每篇文章都提到了它，所以我也想提一下！额外的知识很少是坏事。

## 注2：历史课

古老的操作系统，包括经典的Mac OS和NT之前很久的Windows版本，使用抢占式多任务的前身。操作系统不决定何时抢占程序，而是程序自己选择让位给操作系统。它们会触发一个软件中断说，“嘿，

你现在可以让另一个程序运行了。“这些显式的让步是操作系统重新获得控制权并切换到下一个调度进程的唯一方式。

这被称为 [协作多任务](#)。它有几个主要缺陷：恶意或设计不良的程序很容易冻结整个操作系统，而且几乎不可能确保实时/时间敏感任务的时间一致性。由于这些原因，技术界很久以前就转向了抢占式多任务，再也没有回头。

# Chapter 3: How to Run a Program

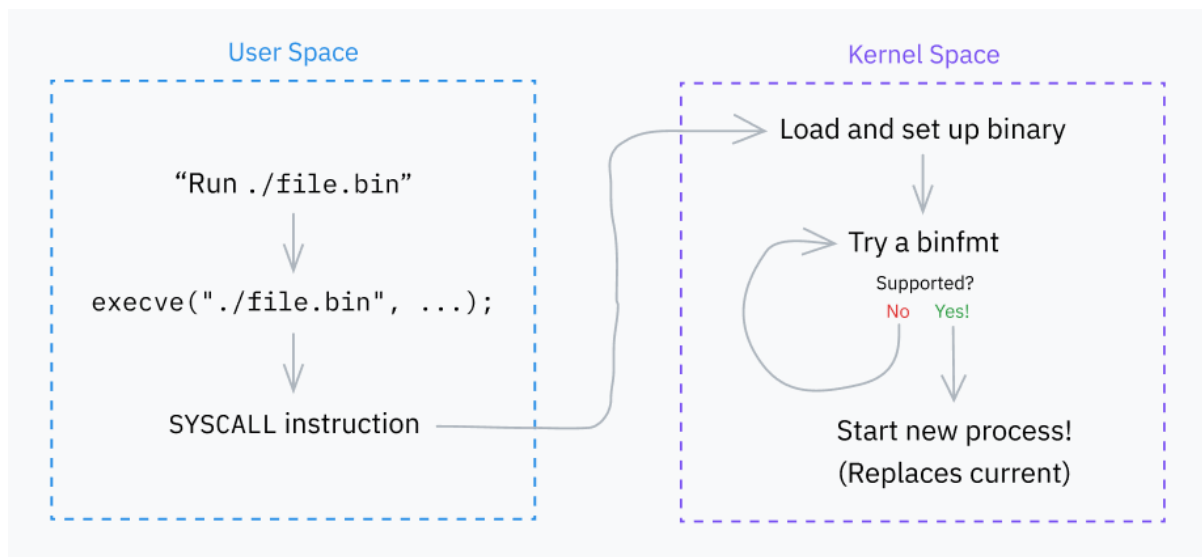
到目前为止，我们已经讨论了CPU如何执行从可执行文件加载的机器码，什么是基于环的安全性，以及系统调用如何工作。在本节中，我们将深入Linux内核，了解程序最初是如何被加载和运行的。

我们将特别关注x86-64架构上的Linux。为什么？

- Linux是一个功能齐全的生产级操作系统，适用于桌面、移动和服务器的用例。Linux是开源的，所以通过阅读其源代码很容易进行研究。我将在本文中直接引用一些内核代码！
- x86-64是大多数现代桌面计算机使用的架构，也是许多代码的目标架构。我提到的特定于x86-64的行为子集将具有很好的通用性。

我们学到的大部分内容将很好地适用于其他操作系统和架构，即使它们在各种具体方面有所不同。

## Exec系统调用的基本行为



让我们从一个非常重要的系统调用开始：`execve`。它加载一个程序，如果成功，就用该程序替换当前进程。还有几个其他的系统调用（`execlp`、`execvpe`等）存在，但它们都以各种方式基于`execve`。

## 题外话：execveat

`execve`实际上是基于`execveat`构建的，`execveat`是一个更通用的系统调用，可以运行带有一些配置选项的程序。为了简单起见，我们主要讨论`execve`；唯一的区别是它为`execveat`提供了一些默认值。

好奇`ve`代表什么？`v`意味着一个参数是参数的向量（列表）（`argv`），`e`意味着另一个参数是环境变量的向量（`envp`）。各种其他`exec`系统调用有不同的后缀来指定不同的调用签名。`execveat`中的`at`就是“at”，因为它指定了运行`execve`的位置。

`execve`的调用签名是：

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- `filename`参数指定要运行的程序的路径。
- `argv`是一个以null结尾（意味着最后一项是空指针）的程序参数列表。你经常会看到传递给C语言`main`函数的`argc`参数实际上是后来由系统调用计算的，因此需要null终止。
- `envp`参数包含另一个以null结尾的环境变量列表，用作应用程序的上下文。它们...按惯例是`KEY=VALUE`对。按惯例。我爱计算机。

有趣的事实！你知道程序的第一个参数是程序名称的那个惯例吗？那纯粹是一个惯例，实际上并不是由`execve`系统调用本身设置的！第一个参数将是传递给`execve`的`argv`参数中的第一项，即使它与程序名称无关。

有趣的是，`execve`确实有一些代码假设`argv[0]`是程序名称。稍后当我们讨论解释型脚本语言时会详细讨论这一点。

## 步骤0：定义

我们已经知道系统调用是如何工作的，但我们从未见过真实世界的代码示例！让我们看看Linux内核的源代码，看看`execve`在底层是如何定义的：

fs/exec.c

```
2105 SYSCALL_DEFINE3(execve,  
2106                 const char __user *, filename,  
2107                 const char __user *const __user *, argv,  
2108                 const char __user *const __user *, envp)  
2109 {  
2110     return do_execve(getname(filename), argv, envp);  
2111 }
```

`SYSCALL_DEFINE3`是定义3参数系统调用代码的宏。

我很好奇为什么`arity`（参数数量）在宏名中是硬编码的；我在网上搜索后了解到，这是为了修复[某个安全漏洞](#)的解决方案。

文件名参数被传递给`getname()`函数，该函数将字符串从用户空间复制到内核空间，并做一些使用跟踪的事情。它返回一个`filename`结构体，定义在`include/linux/fs.h`中。它存储了指向用户空间原始字符串的指针，以及指向复制到内核空间的值的新指针：

include/linux/fs.h

```
2294 struct filename {  
2295     const char          *name; /* pointer to actual string */  
2296     const __user char   *uptr; /* original userland pointer */  
2297     int                 refcnt;  
2298     struct audit_names   *aname;  
2299     const char          iname[];  
2300 };
```

然后`execve`系统调用调用`do_execve()`函数。这又调用`do_execveat_common()`并传入一些默认值。我之前提到的`execveat`系统调用也调用`do_execveat_common()`，但传递更多用户提供的选项。

在下面的代码片段中，我包含了`do_execve`和`do_execveat`的定义：

fs/exec.c

```
2028 static int do_execve(struct filename *filename,
2029                      const char __user *const __argv,
2030                      const char __user *const __envp)
2031 {
2032     struct user_arg_ptr argv = { .ptr.native = __argv };
2033     struct user_arg_ptr envp = { .ptr.native = __envp };
2034     return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
2035 }
2036
2037 static int do_execveat(int fd, struct filename *filename,
2038                      const char __user *const __argv,
2039                      const char __user *const __envp,
2040                      int flags)
2041 {
2042     struct user_arg_ptr argv = { .ptr.native = __argv };
2043     struct user_arg_ptr envp = { .ptr.native = __envp };
2044
2045     return do_execveat_common(fd, filename, argv, envp, flags);
2046 }
```

[格式如原文]

在`execveat`中，文件描述符（指向某种资源的ID类型）被传递给系统调用，然后传递给`do_execveat_common`。这指定了相对于哪个目录执行程序。

在`execve`中，使用了一个特殊值`AT_FDCWD`作为文件描述符参数。这是Linux内核中的一个共享常量，告诉函数将路径名解释为相对于当前工作目录。接受文件描述符的函数通常包括一个手动检查，如`if (fd == AT_FDCWD) { /* 特殊代码路径 */ }`。

## 步骤1：设置

我们现在到达了`do_execveat_common`，这是处理程序执行的核心函数。我们要暂时从盯着代码的状态退一步，获得这个函数做什么的更大图景。

`do_execveat_common`的第一个主要工作是设置一个叫做`linux_binprm`的结构体。我不会包含[整个结构体定义](#)的副本，但有几个重要的字段需要讨论：

- 定义了像`mm_struct`和`vm_area_struct`这样的数据结构，为新程序准备虚拟内存管理。



- 计算并存储`argc`和`envc`以传递给程序。
- `filename`和`interp`分别存储程序的文件名和其解释器。这些开始时是相等的，但在某些情况下可能会改变：一种情况是运行带有`shebang`的解释型脚本。例如，当执行Python程序时，`filename`指向源文件，但`interp`是Python解释器的路径。
- `buf`是一个数组，填充了要执行的文件的前256个字节。它用于检测文件的格式并加载脚本`shebang`。

（今天我学到：`binprm`代表**binary program**。）

让我们仔细看看这个`buf`缓冲区：

```
linux_binprm @ include/linux/binfmts.h
64          char buf[BINPRM_BUF_SIZE];
```

正如我们所见，它的长度定义为常量`BINPRM_BUF_SIZE`。通过在代码库中搜索这个字符串，我们可以在`include/uapi/linux/binfmts.h`中找到它的定义：

```
include/uapi/linux/binfmts.h
18 /* sizeof(linux_binprm->buf) */
19 #define BINPRM_BUF_SIZE 256
```

所以，内核将执行文件的前256个字节加载到这个内存缓冲区中。

### 题外话：什么是UAPI？

你可能注意到上面代码的路径包含`/uapi/`。为什么长度不在与`linux_binprm`结构体相同的文件`include/linux/binfmts.h`中定义？

UAPI代表“用户空间API”。在这种情况下，它意味着有人决定缓冲区的长度应该是内核公共API的一部分。理论上，所有UAPI都对用户空间暴露，而所有非UAPI都是内核代码的私有部分。

内核和用户空间代码最初共存于一个混杂的整体中。2012年，UAPI代码被[重构到一个单独的目录](#)，试图提高可维护性。

## 步骤2 : Binfmts

内核的下一个主要工作是遍历一堆”binfmt”（二进制格式）处理程序。这些处理程序定义在诸如 `fs/binfmt_elf.c` 和 `fs/binfmt_flat.c` 这样的文件中。[内核模块](#) 也可以向池中添加自己的binfmt处理程序。

每个处理程序都暴露一个 `load_binary()` 函数，该函数接受一个 `linux_binprm` 结构体，并检查处理程序是否理解程序的格式。

这通常涉及在缓冲区中查找[魔数](#)，尝试解码程序的开头（也来自缓冲区），和/或检查文件扩展名。如果处理程序支持该格式，它会准备程序执行并返回成功代码。否则，它会提前退出并返回错误代码。

内核尝试每个binfmt的 `load_binary()` 函数，直到找到一个成功的。有时这些会递归运行；例如，如果一个脚本指定了一个解释器，而该解释器本身又是一个脚本，那么层次结构可能是 `binfmt_script > binfmt_script > binfmt_elf`（其中ELF是链的末端的可执行格式）。

## 格式亮点：脚本

在Linux支持的许多格式中，`binfmt_script` 是我想特别谈论的第一个。

你有没有读过或写过[shebang](#)？那个在一些脚本开头指定解释器路径的行？

```
1  #!/bin/bash
```

我一直以为这些是由shell处理的，但并非如此！Shebangs实际上是内核的一个功能，脚本和其他所有程序一样使用相同的系统调用执行。计算机太酷了。

看看 `fs/binfmt_script.c` 是如何检查一个文件是否以shebang开头的：

```
load_script @ fs/binfmt_script.c

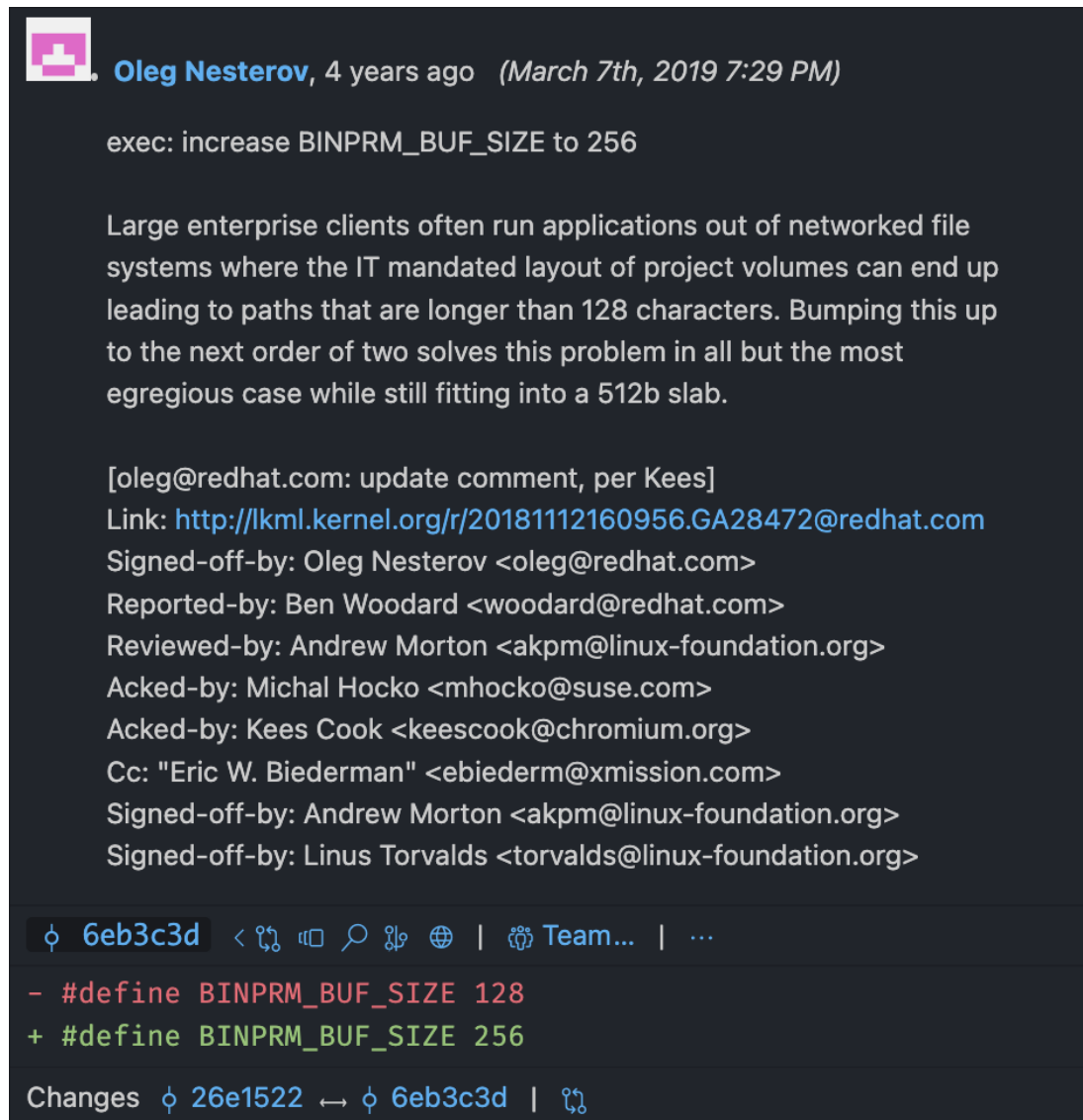
40          /* Not ours to exec if we don't start with "#!". */
41          if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
42              return -ENOEXEC;
```

如果文件确实以shebang开头，binfmt处理程序然后读取解释器路径和路径后的任何空格分隔的参数。它会在遇到换行符或缓冲区结束时停止。

这里有两件有趣的、古怪的事情正在发生。

首先，还记得linux\_binprm中那个填充了文件前256个字节的缓冲区吗？它用于可执行格式检测，但同样的缓冲区也在binfmt\_script中用于读取shebangs。

在我的研究过程中，我读到一篇文章描述这个缓冲区长度为128字节。在那篇文章发布后的某个时候，长度被翻倍到256字节！好奇为什么会这样，我查看了Git blame —— 一个记录谁编辑了某行代码的日志 —— 查看Linux源代码中BINPRM\_BUF\_SIZE定义的那一行。果不其然...



The screenshot shows a Git commit interface. At the top, it says 'Oleg Nesterov, 4 years ago (March 7th, 2019 7:29 PM)'. Below that is the commit message: 'exec: increase BINPRM\_BUF\_SIZE to 256'. The message continues with a paragraph explaining the change: 'Large enterprise clients often run applications out of networked file systems where the IT mandated layout of project volumes can end up leading to paths that are longer than 128 characters. Bumping this up to the next order of two solves this problem in all but the most egregious case while still fitting into a 512b slab.' This is followed by a list of reviewers and their email addresses, including Kees Cook, Andrew Morton, and Linus Torvalds. At the bottom, a diff shows the change from '#define BINPRM\_BUF\_SIZE 128' to '#define BINPRM\_BUF\_SIZE 256'.

```
exec: increase BINPRM_BUF_SIZE to 256

Large enterprise clients often run applications out of networked file
systems where the IT mandated layout of project volumes can end up
leading to paths that are longer than 128 characters. Bumping this up
to the next order of two solves this problem in all but the most
egregious case while still fitting into a 512b slab.

[oleg@redhat.com: update comment, per Kees]
Link: http://lkml.kernel.org/r/20181112160956.GA28472@redhat.com
Signed-off-by: Oleg Nesterov <oleg@redhat.com>
Reported-by: Ben Woodard <woodard@redhat.com>
Reviewed-by: Andrew Morton <akpm@linux-foundation.org>
Acked-by: Michal Hocko <mhocko@suse.com>
Acked-by: Kees Cook <keescook@chromium.org>
Cc: "Eric W. Biederman" <ebiederm@xmission.com>
Signed-off-by: Andrew Morton <akpm@linux-foundation.org>
Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>

φ 6eb3c3d < > | Team... | ...
- #define BINPRM_BUF_SIZE 128
+ #define BINPRM_BUF_SIZE 256

Changes φ 26e1522 ↔ φ 6eb3c3d | >
```

计算机太酷了！

由于shebangs是由内核处理的，并且从buf而不是加载整个文件中获取，它们总是被截断到buf的长度。显然，4年前，有人对内核截断他们超过128个字符的路径感到恼火，他们的解决方案是通过将缓冲区大小翻倍来将截断点翻倍！今天，在你自己的Linux机器上，如果你有一个超过256个字符长的shebang行，超过256个字符的所有内容将完全丢失。

file.bin	
Loaded into buf (first 256 bytes)	43 05 cb 04 97 e4 34 23 34 09 c7 a2 7f 35 a8 89 12 0e fb 79 fe ce 83 64 d1 f3 b4 a2 fb e1 26 0c d8 88 bd 1e 6d c0 9e 38 3a 8c c7 06 59 10 99 c7 20 c8 70 fd d7 09 1b 5a a4 8a 0b c9 74 74 11 30 18 6f c2 56 bf eb 92 51 41 dd 88 76 08 45 51 b3 df 99 f1 ab 40 cf 50 c4 86 65 b8 4a d0 a2 34 f4 99 85 29 06 c9 6e c2 e9 3e 65 ff 28 b1 65 31 39 11 1a 8d c1 89 cd 17 8b 68 16 ed 47 21 5f c9 68 4e 6b 66 cb 07 02 e0 59 22 32 53 55 6e d6 3e 37 0c 59 15 55 e9 40 47 e5 0b 36 52 0d 0f 13 d0 4d cc f0 4c fa 5c 8f 4a 2e 7f bd b5 ed 22 9a ce 6c 40 46 30 8e bc 6e cb fd 27 3a 17 ac 1c 41 f3 66 02 4c 2f e0 00 7f 5a 1e f4 e4 13 23 05 8c 39 f1 a0 d0 48 68 27 c6 8b 96 9d 8b 54 f8 5f 63 75 29 ef 39 54 16 72 6e fe 9e b3 a6 27 4d ef 3c 46 54 e2 27 85 3a bb 65 45 cd 63 89 b5 a4 a9 ba 07 ea
Ignored (past 256 bytes)	21 1e 54 e5 e0 1f 2a 93 e8 25 53 00 b0 d7 58 bc f6 64 85 95 e6 3e 53 a4 54 97 d0 f9 fd 70 f5 14 ce 66 7a 75 44 df 5c d4 b2 16 d3 cd 46 2c 8e a2 24 47 12 65 25 bf fa 9f a9 18 1c 02 49 49 23 d1

想象一下因为这个而出现bug。想象一下试图找出破坏你代码的根本原因。想象一下，当你发现问题深藏在Linux内核中时的感受。可怜那些在大型企业中发现路径的一部分神秘消失的下一个IT人员。

**\*\*第二件古怪的事：\*\***还记得argv[0]是程序名称只是一个惯例吗，调用者可以传递任何他们想要的argv给exec系统调用，它会不加修改地通过？

恰好binfmt\_script是那些假设argv[0]是程序名称的地方之一。它总是移除argv[0]，然后在argv的开头添加以下内容：

- 解释器的路径
- 解释器的参数
- 脚本的文件名

## 示例：参数修改

让我们看一个`execve`调用的示例：

```
// 参数：filename, argv, envp
execve("./script", [ "A", "B", "C" ], []);
```

这个假设的`script`文件的第一行有以下shebang：

```
script
1  #!/usr/bin/node --experimental-module
```

最终传递给Node解释器的修改后的`argv`将是：

```
[ "/usr/bin/node", "--experimental-module", "./script", "B", "C" ]
```

更新`argv`后，处理程序通过将`linux_binprm.interp`设置为解释器路径（在本例中是Node二进制文件）来完成准备文件执行的工作。最后，它返回0表示成功准备程序执行。

## 格式亮点：杂项解释器

另一个有趣的处理程序是`binfmt_misc`。它开放了通过用户空间配置添加一些有限格式的能力，方法是在`/proc/sys/fs/binfmt_misc/`挂载一个特殊的文件系统。程序可以对这个目录中的文件执行[特殊格式的](#)写入来添加自己的处理程序。每个配置条目指定：

- 如何检测他们的文件格式。这可以指定在某个偏移处的魔数，或要查找的文件扩展名。
- 解释器可执行文件的路径。没有办法指定解释器参数，所以如果需要这些，就需要一个包装脚本。
- 一些配置标志，包括一个指定`binfmt_misc`如何更新`argv`的标志。

这个`binfmt_misc`系统经常被Java安装使用，配置为通过它们的`0xCAFEBAFE`魔数检测类文件，通过它们的扩展名检测JAR文件。在我的特定系统上，配置了一个处理程序，通过`.pyc`扩展名检测Python字

节码并将其传递给适当的处理程序。

这是一种让程序安装程序添加对自己格式的支持的很酷的方式，而不需要编写高度特权的内核代码。

## 最后（不是林肯公园的歌）

一个exec系统调用总会最终走向两条路径之一：

- 它最终会达到一个它理解的可执行二进制格式，可能在经过几层脚本解释器之后，并运行那个代码。在这一点上，旧代码已被替换。
- ... 或者它会用尽所有选项，夹着尾巴返回错误代码给调用程序。

如果你曾经使用过类Unix系统，你可能注意到从终端运行的shell脚本即使没有shebang行或`.sh`扩展名也能执行。如果你有一个非Windows终端，你现在就可以测试这个：

Shell session

```
$ echo "echo hello" > ./file
$ chmod +x ./file
$ ./file
hello
```

（`chmod +x`告诉OS一个文件是可执行的。否则你将无法运行它。）

那么，为什么shell脚本会作为shell脚本运行呢？内核的格式处理程序应该没有明确的方法来检测没有任何可辨识标签的shell脚本！

嗯，事实证明这个行为不是内核的一部分。它实际上是你的`shell`处理失败情况的一种常见方式。

当你使用shell执行一个文件，而exec系统调用失败时，大多数shell会尝试将文件作为`shell`脚本重新执行，方法是执行一个shell，将文件名作为第一个参数。Bash通常会使用自己作为这个解释器，而ZSH使用whatever `sh`是，通常是[Bourne shell](#)。

这种行为如此普遍，是因为它在[POSIX](#)中被指定，POSIX是一个旧标准，旨在使代码在Unix系统之间可移植。虽然大多数工具或操作系统并不严格遵循POSIX，但它的许多约定仍然被共享。

如果[一个exec系统调用]由于等同于[ENOEXEC]错误的错误而失败，**shell应执行等同于以命令名作为其第一个操作数调用shell的命令**，将任何剩余参数传递给新shell。如果可执行文件不是文本文件，shell可能会绕过这个命令执行。在这种情况下，它应该写一个错误消息，并返回126的退出状态。

来源：[\*Shell命令语言, POSIX.1-2017\*](#)

计算机太酷了！



## Chapter 4: Becoming an Elf-Lord

我们现在对 `execve` 有了相当全面的理解。在大多数路径的最后,内核会达到一个包含要启动的机器代码的最终程序。通常,在实际跳转到代码之前需要一个设置过程 - 例如,程序的不同部分必须加载到内存中的正确位置。每个程序需要不同数量的内存用于不同的事情,所以我们有标准的文件格式来指定如何设置程序以执行。虽然 Linux 支持许多这样的格式,但迄今为止最常见的格式是 *ELF* (可执行和可链接格式)。



(感谢 [Nicky Case](#) 提供的可爱插图。)

### 题外话:精灵无处不在?

当你在 Linux 上运行应用程序或命令程序时,极有可能它是一个 ELF 二进制文件。然而,在 macOS 上,事实上的标准格式是 [Mach-O](#)。Mach-O 做的事情和 ELF 一样,但结构不同。在 Windows 上,.exe 文件使用 [可移植可执行文件](#) 格式,这又是一个概念相同但格式不同的格式。



在 Linux 内核中,ELF 二进制文件由 `binfmt_elf` 处理程序处理,它比许多其他处理程序更复杂,包含数千行代码。它负责从 ELF 文件中解析出某些细节,并使用它们将进程加载到内存中并执行。

我运行了一些命令行操作来按行数对 `binfmt` 处理程序进行排序:

Shell session

```
$ wc -l binfmt_* | sort -nr | sed 1d
2181 binfmt_elf.c
1658 binfmt_elf_fdpic.c
944 binfmt_flat.c
836 binfmt_misc.c
158 binfmt_script.c
64 binfmt_elf_test.c
```

## 文件结构

在更深入地研究 `binfmt_elf` 如何执行 ELF 文件之前,让我们看看文件格式本身。ELF 文件通常由四个部分组成:

### Structure of an ELF File

#### ELF Header

Basic information about the binary, and locations of PHT and SHT.

#### Program Header Table (PHT)

Describes how and where to load the ELF file's data into memory.

#### Section Header Table (SHT)

Optional “map” of the data to assist in debugging.

#### Data

All of the binary's data. The PHT and SHT point into this section.

## ELF 头

每个 ELF 文件都有一个 [ELF 头](#)。它有非常重要的任务,传达关于二进制文件的基本信息,例如:

- 它设计运行的处理器。ELF 文件可以包含不同处理器类型的机器代码,如 ARM 和 x86。
- 该二进制文件是否旨在作为可执行文件独立运行,或者是否旨在作为“动态链接库”被其他程序加载。我们很快会详细讨论动态链接是什么。
- 可执行文件的入口点。后面的部分会详细说明将 ELF 文件中包含的数据加载到内存中的确切位置。入口点是一个内存地址,指向整个进程加载后第一条机器代码指令在内存中的位置。

ELF 头总是在文件的开头。它指定了程序头表和段头的位置,这些可以在文件中的任何位置。这些表反过来指向存储在文件其他地方的数据。

## 程序头表

[程序头表](#) 是一系列条目,包含有关如何在运行时加载和执行二进制文件的具体细节。每个条目都有一个类型字段,说明它指定的是什么细节 - 例如,`PT_LOAD` 意味着它包含应该加载到内存中的数据,而 `PT_NOTE` 意味着该段包含不一定要加载到任何地方的信息性文本。

### Common Program Header Types

<b>PT_LOAD</b>	Data to be loaded into memory.
<b>PT_NOTE</b>	Freeform text like copyright notices, version info, etc.
<b>PT_DYNAMIC</b>	Info about dynamic linking.
<b>PT_INTERP</b>	Path to the location of an “ELF interpreter.”

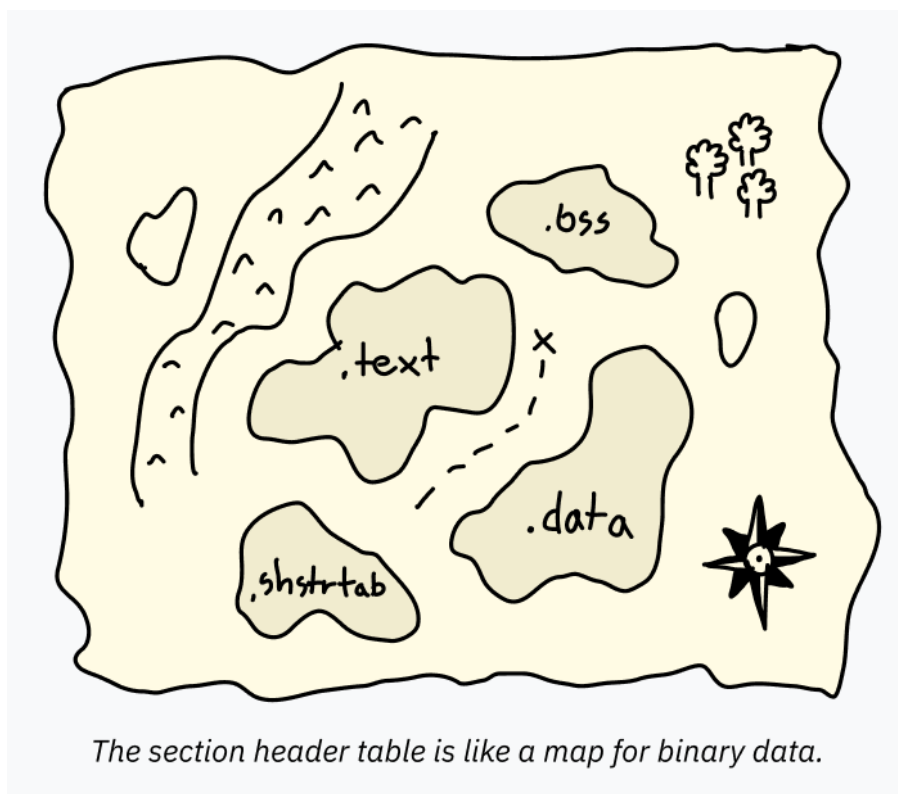
每个条目指定有关其数据在文件中的位置以及有时如何将数据加载到内存中的信息:

- 它指向其数据在 ELF 文件中的位置。
- 它可以指定数据应该加载到内存中的虚拟内存地址。如果该段不打算加载到内存中,这通常留空。
- 两个字段指定数据的长度:一个用于文件中数据的长度,一个用于要创建的内存区域的长度。如果内存区域长度大于文件中的长度,多余的内存将用零填充。这对于可能想在运行时使用静态内存段的程序有益;这些空的内存段通常称为 [BSS](#) 段。

- 最后,一个标志字段指定如果它被加载到内存中应该允许什么操作:`PF_R` 使其可读,`PF_W` 使其可写,`PF_X` 意味着它是应该允许在 CPU 上执行的代码。

## 段头表

[段头表](#) 是一系列包含有关段信息的条目。这个段信息就像一张地图,绘制了 ELF 文件内部的数据。它使得[像调试器这样的程序](#)很容易理解数据不同部分的预期用途。



例如,程序头表可以指定一大块数据一起加载到内存中。那个单一的 `PT_LOAD` 块可能同时包含代码和全局变量!运行程序不需要单独指定这些;CPU 只是从入口点开始,向前步进,在程序请求时和请求的地方访问数据。然而,用于分析程序的软件(如调试器)需要确切知道每个区域的开始和结束位置,否则它可能会尝试将一些说“hello”的文本解码为代码(由于这不是有效的代码,会导致崩溃)。这些信息存储在段头表中。

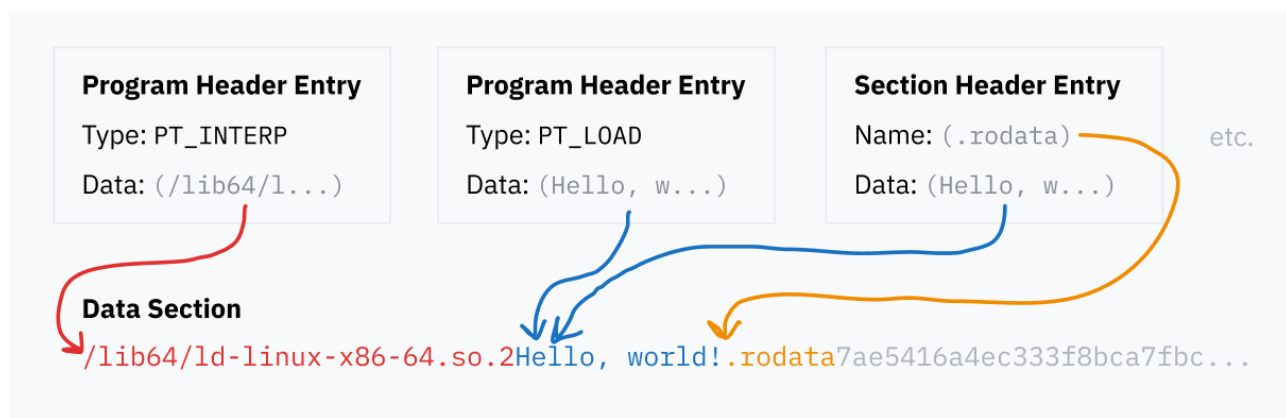
虽然通常包括,但段头表实际上是可选的。ELF 文件可以在完全移除段头表的情况下完美运行,想要隐藏其代码作用的开发者有时会故意从其 ELF 二进制文件中剥离或混淆段头表,以[使它们更难解码](#)。

每个段都有一个名称、一个类型和一些标志,指定它应该如何使用和解码。标准名称通常按惯例以点开头。最常见的段是:

- `.text`: 要加载到内存中并在 CPU 上执行的机器代码。类型为 `SHT_PROGBITS`, 带有 `SHF_EXECINSTR` 标志标记为可执行, 以及 `SHF_ALLOC` 标志, 表示它被加载到内存中执行。(不要被名称混淆, 它仍然只是二进制机器代码! 我一直觉得它被称为 `.text` 有点奇怪, 尽管它不是可读的“文本”。)
- `.data`: 要加载到内存中的可执行文件中硬编码的初始化数据。例如, 包含一些文本的全局变量可能在这个段中。如果你编写低级代码, 这是静态变量所在的段。这也是 `SHT_PROGBITS` 类型, 这只是意味着该段包含“程序的信息”。它的标志是 `SHF_ALLOC` 和 `SHF_WRITE`, 将其标记为可写内存。
- `.bss`: 我之前提到过, 通常会有一些分配的内存从零开始。在 ELF 文件中包含一堆空字节会是一种浪费, 所以使用了一种特殊的段类型叫做 BSS。在调试过程中了解 BSS 段很有帮助, 所以还有一个段头表条目指定要分配的内存长度。它的类型是 `SHT_NOBITS`, 标志为 `SHF_ALLOC` 和 `SHF_WRITE`。
- `.rodata`: 这和 `.data` 类似, 只是它是只读的。在一个非常基础的运行 `printf("Hello, world!")` 的 C 程序中, “Hello world!” 字符串会在 `.rodata` 段中, 而实际的打印代码会在 `.text` 段中。
- `.shstrtab`: 这是一个有趣的实现细节! 段的名称本身(如 `.text` 和 `.shstrtab`) 并不直接包含在段头表中。相反, 每个条目包含一个偏移量, 指向 ELF 文件中包含其名称的位置。这样, 段头表中的每个条目可以是相同的大小, 使它们更容易解析 - 名称的偏移量是一个固定大小的数字, 而在表中包含名称会使用可变大小的字符串。所有这些名称数据都存储在自己的段中, 称为 `.shstrtab`, 类型为 `SHT_STRTAB`。

## 数据

程序和段头表条目都指向 ELF 文件内的数据块, 无论是将它们加载到内存中, 还是指定程序代码的位置, 或者仅仅是命名段。所有这些不同的数据片段都包含在 ELF 文件的数据部分中。



## 链接的简要解释

回到 `binfmt_elf` 代码:内核关心程序头表中的两种类型的条目。

`PT_LOAD` 段指定所有程序数据(如 `.text` 和 `.data` 段)需要加载到内存中的位置。内核从 ELF 文件中读取这些条目,将数据加载到内存中,以便 CPU 可以执行程序。

内核关心的另一种程序头表条目类型是 `PT_INTERP`,它指定了一个“动态链接运行时”。

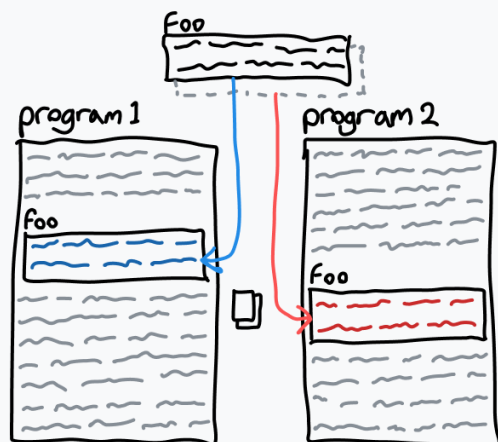
在我们讨论动态链接是什么之前,让我们先谈谈一般的“链接”。程序员倾向于在可重用代码库的基础上构建他们的程序 - 例如,我们之前讨论过的 `libc`。在将源代码转换为可执行二进制文件时,一个名为链接器的程序通过查找库代码并将其复制到二进制文件中来解决所有这些引用。这个过程被称为**静态链接**,意味着外部代码直接包含在分发的文件中。

然而,有些库非常常见。你会发现基本上每个程序都使用 `libc`,因为它是与操作系统通过系统调用交互的标准接口。在计算机上的每个程序中都包含一个单独的 `libc` 副本将是对空间的糟糕使用。此外,如果库中的 bug 能在一个地方修复,而不是等待使用该库的每个程序更新,那会很好。动态链接是解决这些问题的方案。

如果一个静态链接的程序需要一个名为 `bar` 的库中的函数 `foo`,该程序将包含整个 `foo` 的副本。然而,如果是动态链接的,它只会包含一个引用,说“我需要 `bar` 库中的 `foo`”。当程序运行时,希望 `bar` 安装在计算机上,`foo` 函数的机器代码可以按需加载到内存中。如果计算机上安装的 `bar` 库被更新,新代码将在程序下次运行时加载,而不需要程序本身做任何更改。

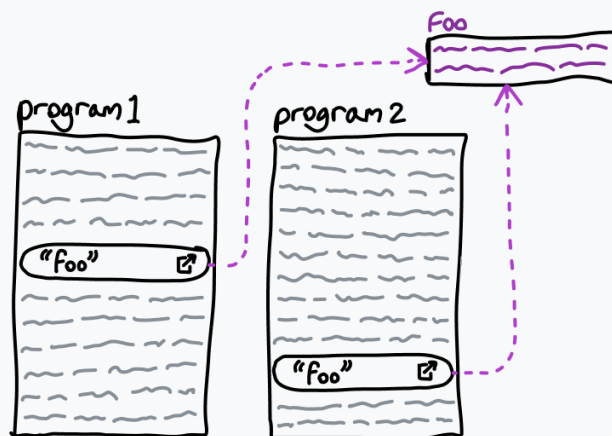
## Static Linking

Library functions are **copied from the developer's computer** into each binary at build time.



## Dynamic Linking

Binaries reference the names of library functions, which are **loaded from the user's computer** at runtime.



## 实际中的动态链接

在 Linux 上,像 `bar` 这样的可动态链接库通常被打包成带有 `.so` (共享对象)扩展名的文件。这些 `.so` 文件就像程序一样也是 ELF 文件 - 你可能还记得 ELF 头包含一个字段,用于指定文件是可执行文件还是库。此外,共享对象在段头表中有一个 `.dynsym` 段,其中包含有关从文件导出并可以动态链接到的符号的信息。

在 Windows 上,像 `bar` 这样的库被打包成 `.dll` (**d**ynamic **l**ink **l**ibrary) 文件。macOS 使用 `.dylib` (**d**ynamically linked **l**ibrary) 扩展名。就像 macOS 应用程序和 Windows `.exe` 文件一样,这些格式与 ELF 文件略有不同,但概念和技术是相同的。

两种链接类型之间的一个有趣区别是,使用静态链接时,只有使用的库部分会包含在可执行文件中并加载到内存中。而使用动态链接时,整个库都会加载到内存中。这最初可能听起来效率较低,但实际上它允许现代操作系统通过将库加载到内存中一次,然后在进程之间共享该代码来节省更多空间。只有代码可以共享,因为库需要为不同的程序保持不同的状态,但节省的空间仍然可能在数十到数百兆字节 RAM 的数量级。

# 执行

让我们回到内核运行 ELF 文件的话题:如果它正在执行的二进制文件是动态链接的,操作系统不能直接跳转到二进制文件的代码,因为会有缺失的代码 - 记住,动态链接的程序只有对它们需要的库函数的引用!

要运行二进制文件,操作系统需要弄清楚需要哪些库,加载它们,用实际的跳转指令替换所有命名指针,然后才开始实际的程序代码。这是非常复杂的代码,与 ELF 格式深度交互,所以它通常是一个独立的程序,而不是内核的一部分。ELF 文件在程序头表的 `PT_INTERP` 条目中指定它们想要使用的程序的路径(通常是类似 `/lib64/ld-linux-x86-64.so.2` 这样的东西)。

在读取 ELF 头并扫描程序头表后,内核可以为新程序设置内存结构。它首先将所有 `PT_LOAD` 段加载到内存中,填充程序的静态数据、BSS 空间和机器代码。如果程序是动态链接的,内核将不得不执行 [ELF 解释器](#)(`PT_INTERP`),所以它也将解释器的数据、BSS 和代码加载到内存中。

现在内核需要设置 CPU 在返回用户空间时要恢复的指令指针。如果可执行文件是动态链接的,内核将指令指针设置为内存中 ELF 解释器代码的开始。否则,内核将其设置为可执行文件的开始。

内核几乎准备好从系统调用返回了(记住,我们仍然在 `execve` 中)。它将 `argc`、`argv` 和环境变量推送到栈上,以便程序在开始时读取。

现在寄存器被清除。在处理系统调用之前,内核将寄存器的当前值存储到栈中,以便在切换回用户空间时恢复。在返回用户空间之前,内核将栈的这部分清零。

最后,系统调用结束,内核返回用户空间。它恢复寄存器,现在寄存器已被清零,并跳转到存储的指令指针。该指令指针现在是新程序(或 ELF 解释器)的起点,当前进程已被替换!



# Chapter 5: The Translator in Your Computer

直到现在,我每次谈到读写内存时都有点模糊。例如,ELF 文件指定了将数据加载到的特定内存地址,那为什么不同进程试图使用冲突的内存时不会出现问题?为什么每个进程似乎都有不同的内存环境?

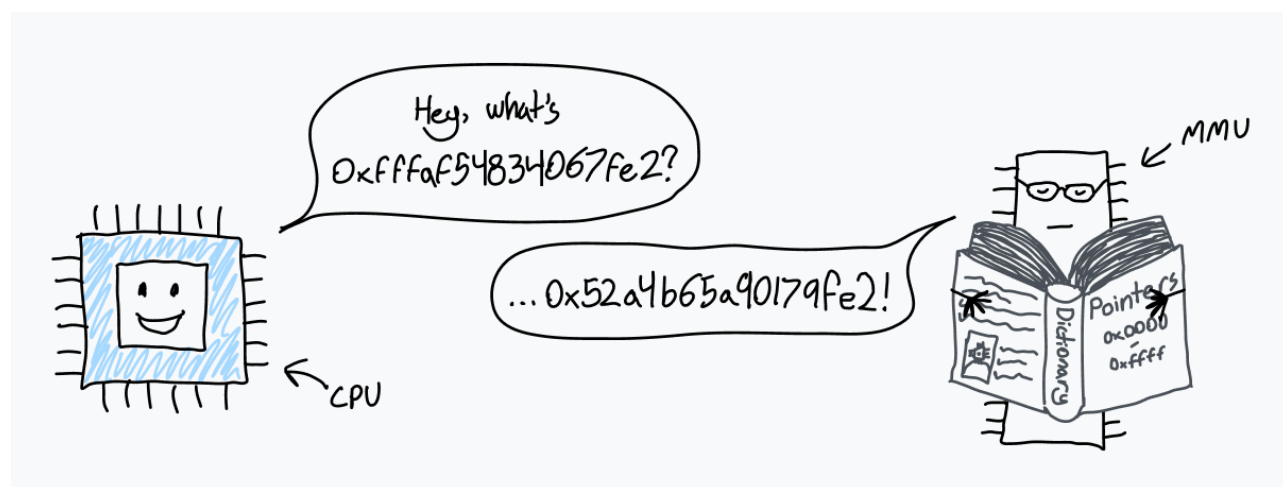
此外,我们究竟是如何到达这里的?我们理解 `execve` 是一个将当前进程替换为新程序的系统调用,但这并不能解释如何启动多个进程。它绝对无法解释第一个程序是如何运行的 - 是哪只鸡(进程)下(生成)了所有其他的蛋(其他进程)?

我们正接近旅程的终点。在回答了这两个问题之后,我们将对你的计算机如何从启动到运行你现在正在使用的软件有一个基本完整的理解。

## 内存是假的

所以...关于内存。事实证明,当 CPU 从内存地址读取或写入时,它实际上并不是指物理内存(RAM)中的那个位置。相反,它指向的是虚拟内存空间中的一个位置。

CPU 与一个叫做[内存管理单元](#)(MMU)的芯片通信。MMU 的工作就像一个带字典的翻译员,将虚拟内存中的位置翻译成 RAM 中的位置。当 CPU 接收到从内存地址 `0xffffaf54834067fe2` 读取的指令时,它会要求 MMU 翻译该地址。MMU 在字典中查找,发现匹配的物理地址是 `0x52a4b65a90179fe2`,并将该数字发送回 CPU。然后 CPU 就可以从 RAM 中的那个地址读取了。



当计算机首次启动时,内存访问直接去到物理 RAM。在启动后立即,操作系统创建翻译字典并告诉 CPU 开始使用 MMU。



这个字典实际上被称为页表,这个翻译每次内存访问的系统被称为分页。页表中的条目被称为页,每一页表示虚拟内存中某个块如何映射到 RAM。这些块总是固定大小的,每种处理器架构都有不同的页面大小。x86-64 的默认页面大小为 4 KiB,这意味着每个页面指定了一个 4,096 字节长的内存块的映射。

换句话说,使用 4 KiB 页面时,地址的底部 12 位在 MMU 翻译前后总是相同的 - 12,因为这是索引翻译后得到的 4,096 字节页面所需的位数。

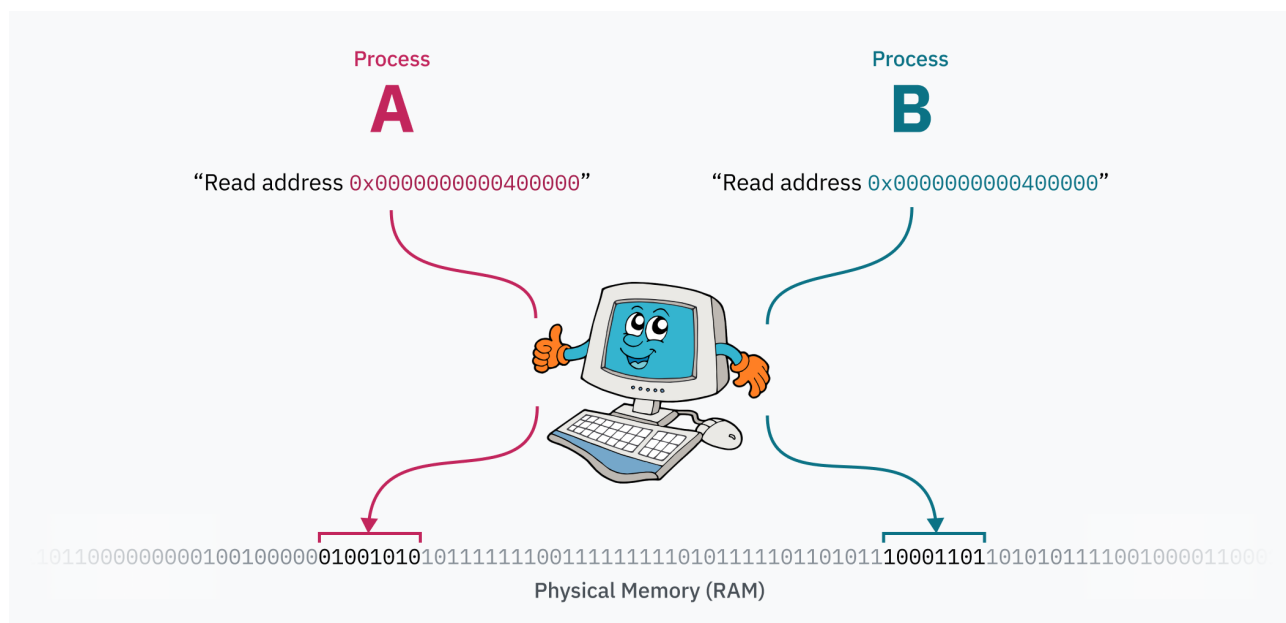
x86-64 还允许操作系统启用更大的 2 MiB 或 4 GiB 页面,这可以提高地址翻译速度,但会增加内存碎片和浪费。页面大小越大,MMU 翻译的地址部分就越小。



页表本身就存在于 RAM 中。虽然它可以包含数百万个条目,但每个条目的大小仅为几个字节,所以页表不会占用太多空间。

要在启动时启用分页,内核首先在 RAM 中构建页表。然后,它将页表开始的物理地址存储在一个称为页表基址寄存器(PTBR)的寄存器中。最后,内核启用分页,用 MMU 翻译所有内存访问。在 x86-64 上,控制寄存器 3(CR3)的前 20 位作为 PTBR。CR0 的第 31 位,指定为 PG(分页),设置为 1 以启用分页。

分页系统的魔力在于可以在计算机运行时编辑页表。这就是每个进程如何拥有自己的独立内存空间 - 当操作系统从一个进程切换到另一个进程时,一个重要的任务是将虚拟内存空间重新映射到物理内存中的不同区域。假设你有两个进程:进程 A 可以在 0x0000000000400000 处访问其代码和数据(可能是从 ELF 文件加载的!),进程 B 可以从完全相同的地址访问其代码和数据。这两个进程甚至可以是同一程序的实例,因为它们实际上并没有争夺那个地址范围!进程 A 的数据在物理内存中离进程 B 很远,当切换到该进程时,内核将其映射到 0x0000000000400000。



### 题外话:诅咒般的 ELF 事实

在某些情况下,`binfmt_elf` 必须将内存的第一页映射为零。一些为 UNIX System V Release 4.0(SVr4)编写的程序,这是 1988 年第一个支持 ELF 的操作系统,依赖于空指针的可读性。而且不知何故,一些程序仍然依赖于那种行为。

看起来实现这一点的 Linux 内核开发者[有点不满](#):

“为什么这样,你问??? 好吧,SVr4 将第 0 页映射为只读,一些应用程序’依赖’于这种行为。由于我们没有能力重新编译这些,我们模拟 SVr4 的行为。叹气。”

叹气。

## 使用分页实现安全

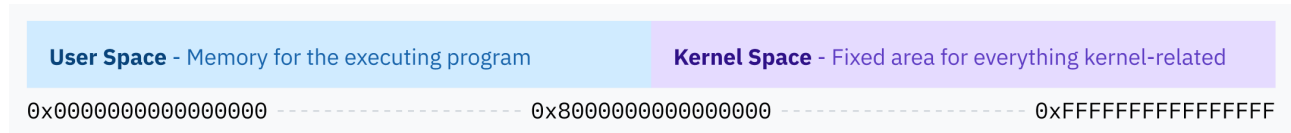
内存分页实现的进程隔离改善了代码的人体工程学(进程不需要意识到其他进程就可以使用内存),但它也创造了一个安全层次:进程无法访问其他进程的内存。这部分回答了本文开头提出的一个原始问题:

如果程序直接在 CPU 上运行,而 CPU 可以直接访问 RAM,为什么代码不能访问其他进程的内存,或者天啊,内核的内存?

还记得吗?感觉已经过去很久了...

那内核内存呢?首先:内核显然需要存储大量自己的数据来跟踪所有正在运行的进程,甚至是页表本身。每次触发硬件中断、软件中断或系统调用,CPU 进入内核模式时,内核代码都需要以某种方式访问那个内存。

Linux 的解决方案是始终将虚拟内存空间的上半部分分配给内核,所以 Linux 被称为[高半内核](#)。Windows 采用了[类似](#)的技术,而 macOS 则...[稍微更复杂](#),阅读它让我的大脑从耳朵里流了出来。~  
(++)~



如果用户空间进程可以读写内核内存,那将是安全性的灾难,所以分页启用了第二层安全:每个页面必须指定权限标志。一个标志决定该区域是可写的还是只读的。另一个标志告诉 CPU 只有内核模式才允许访问该区域的内存。后一个标志用于保护整个高半内核空间 - 整个内核内存空间实际上在用户空间程序的虚拟内存映射中可用,它们只是没有权限访问它。

### Page Table Entry

Present: true

Read/write: read only

User/kernel: all modes

Dirty: false

Accessed: true

etc.

页表本身实际上包含在内核内存空间中!当定时器芯片触发进程切换的硬件中断时,CPU 将特权级别切换到内核模式并跳转到 Linux 内核代码。处于内核模式(Intel ring 0)允许 CPU 访问受内核保护的内存区域。然后内核可以写入页表(位于内存上半部分的某个地方)以重新映射新进程的虚拟内存下半部分。当内核切换到新进程并且 CPU 进入用户模式时,它不能再访问任何内核内存。

几乎每次内存访问都要经过 MMU。中断描述符表处理程序指针?那些也寻址内核的虚拟内存空间。

## 分层分页和其他优化

64 位系统的内存地址长 64 位,这意味着 64 位虚拟内存空间的大小高达 16 [艾字节](#)。这是非常大的,远远大于现在或不久的将来存在的任何计算机。据我所知,任何计算机中最大的 RAM 是在[蓝水超级计算机](#)中,有超过 1.5 拍字节的 RAM。这仍然不到 16 EiB 的 0.01%。

如果虚拟内存空间的每个 4 KiB 部分都需要页表中的一个条目,你将需要 4,503,599,627,370,496 个页表条目。如果页表条目长 8 字节,你将需要 32 皮字节的 RAM 仅用于存储页表。你可能注意到这仍然大于世界纪录的计算机 RAM 最大值。

### 题外话:为什么用这些奇怪的单位?

我知道这很不常见而且真的很难看,但我认为清楚地区分二进制字节大小单位(2 的幂)和公制单位(10 的幂)很重要。千字节,kB,是一个 SI 单位,表示 1,000 字节。kibibyte,KiB,是 IEC 推荐的单位,表示 1,024 字节。就 CPU 和内存地址而言,字节计数通常是 2 的幂,因为计算机是二进制系统。使用 KB(或更糟的 kB)来表示 1,024 会更加模糊。

由于不可能(或至少非常不切实际)对整个可能的虚拟内存空间有连续的页表条目,CPU 架构实现了分层分页。在分层分页系统中,有多个级别的页表,粒度越来越小。顶级条目覆盖大块内存,并指向更小块的页表,创建一个树结构。4 KiB 或任何页面大小的单个块的条目是树的叶子。

x86-64 历史上使用 4 级分层分页。在这个系统中,每个页表条目都是通过将包含表的开始偏移地址的一部分来找到的。这部分从最高有效位开始,作为前缀,使得条目覆盖所有以这些位开始的地址。该条目指向下一级表的开始,该表包含该内存块的子树,再次用下一组位进行索引。

x86-64 的 4 级分页设计者还选择忽略所有虚拟指针的前 16 位以节省页表空间。48 位可以得到 128 TiB 的虚拟地址空间,这被认为足够大了。(完整的 64 位会得到 16 EiB,这有点太多了。)

由于跳过了前 16 位,所以用于索引页表第一级的“最高有效位”实际上从第 47 位而不是第 63 位开始。这也意味着本章前面的高半内核图在技术上是不准确的;内核空间的起始地址应该被描绘为小于 64 位的地址空间的中点。

## x86-64 Paging (4-Level)



分层分页解决了空间问题,因为在树的任何级别,指向下一个条目的指针都可以为空(0x0)。这允许省略页表的整个子树,意味着虚拟内存空间的未映射区域不会占用 RAM 中的任何空间。未映射内存地址的查找可以快速失败,因为 CPU 一旦在树的更高层看到空条目就可以报错。页表条目还有一个存在标志,即使地址看起来有效,也可以用它来标记它们为不可用。

分层分页的另一个好处是能够高效地切换虚拟内存空间的大块。对于一个进程,大片虚拟内存可能映射到物理内存的一个区域,而对于另一个进程,则映射到不同的区域。内核可以将两种映射都存储在内存中,并在切换进程时简单地更新树顶层的指针。如果整个内存空间映射存储为条目的平面数组,内核将不得不更新大量条目,这将很慢,而且仍然需要独立地跟踪每个进程的内存映射。

我说 x86-64 “历史上”使用 4 级分页是因为最近的处理器实现了[5 级分页](#)。5 级分页增加了另一级间接寻址,以及 9 个更多的寻址位,将地址空间扩展到 128 PiB,使用 57 位地址。5 级分页得到了包括 Linux [自 2017 年以来](#)以及最近的 Windows 10 和 11 服务器版本在内的操作系统的支持。

### 题外话:物理地址空间限制

正如操作系统不使用全部 64 位作为虚拟地址,处理器也不使用整个 64 位物理地址。当 4 级分页是标准时,x86-64 CPU 不使用超过 46 位,这意味着物理地址空间仅限于 64 TiB。有了 5 级分页,支持扩展到了 52 位,支持 4 PiB 的物理地址空间。

在操作系统级别,虚拟地址空间大于物理地址空间是有利的。正如 Linus Torvalds [所说](#),“它需要更大,至少要大两倍,而这实际上已经很勉强了,你最好有十倍或更多。任何不明白这一点的人都是个白痴。讨论结束。”

## 交换和按需分页

内存访问可能因为几个原因而失败:地址可能超出范围,可能没有被页表映射,或者可能有一个被标记为不存在的条目。在这些情况下,MMU 将触发一个称为**页错误**的硬件中断,让内核处理这个问题。

在某些情况下,读取确实是无效或被禁止的。在这些情况下,内核可能会以**段错误**错误终止程序。

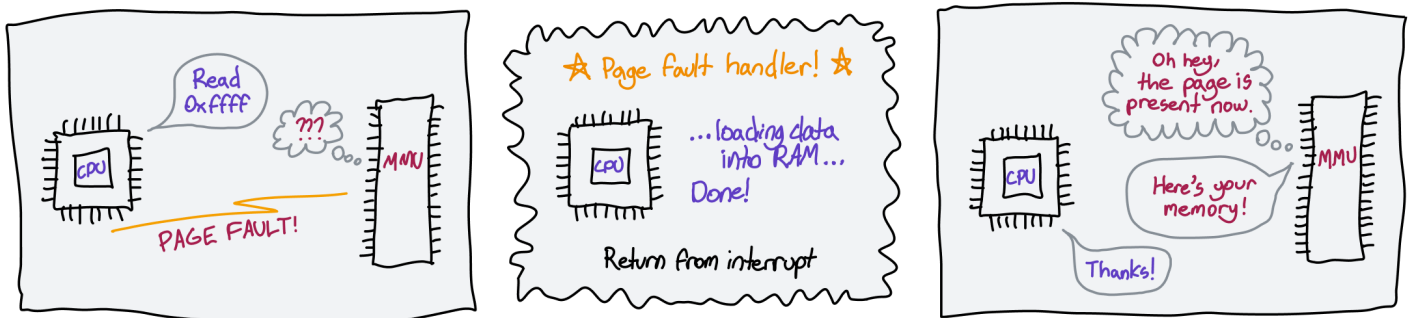
Shell session

```
$ ./program
Segmentation fault (core dumped)
$
```

### 题外话:段错误的本体论

“段错误”在不同的上下文中意味着不同的东西。当内存在没有权限的情况下被读取时,MMU 触发一个称为“段错误”的硬件中断,但“段错误”也是操作系统可以发送给正在运行的程序的信号的名称,用于因任何非法内存访问而终止它们。

在其他情况下,内存访问可以有意失败,允许操作系统填充内存,然后将控制权交还给 CPU 重试。例如,操作系统可以将磁盘上的文件映射到虚拟内存,而不实际将其加载到 RAM 中,然后在请求地址并发生页错误时将其加载到物理内存中。这被称为**按需分页**。



首先,这允许像 [mmap](#) 这样的系统调用存在,它可以懒惰地将整个文件从磁盘映射到虚拟内存。如果你熟悉 LLaMa.cpp,一个泄露的 Facebook 语言模型的运行时,Justine Tunney 最近通过[使所有加载逻辑使用 mmap](#) 显著优化了它。(如果你以前没听说过她,[看看她的东西](#)!Cosmopolitan Libc 和 APE 真的很酷,如果你一直在享受这篇文章的话,可能会觉得有趣。)

显然关于 Justine 参与这个变更有[很多戏剧](#)。我只是指出这一点,这样我就不会被随机的互联网用户大喊大叫。我必须承认我没有读过大部分戏剧,我说的关于 Justine 的东西很酷仍然是非常正确的。

当你执行一个程序及其库时,内核实际上并不将任何东西加载到内存中。它只创建文件的 mmap - 当 CPU 试图执行代码时,页面立即发生错误,内核用一个真正的内存块替换页面。

按需分页还启用了你可能在“交换”或“分页”名下看到的技术。操作系统可以通过将内存页写入磁盘然后从物理内存中移除它们来释放物理内存,但将它们保留在虚拟内存中,同时将存在标志设置为 0。如果读取该虚拟内存,操作系统可以从磁盘将内存恢复到 RAM,并将存在标志设回 1。操作系统可能必须交换出不同的 RAM 部分,为正在从磁盘加载的内存腾出空间。磁盘读写很慢,所以操作系统尝试通过[高效的页面替换算法](#)使交换尽可能少发生。

一个有趣的技巧是使用页表物理内存指针来存储文件在物理存储中的位置。由于 MMU 一看到负的存在标志就会引发页错误,所以它们是无效的内存地址并不重要。这在所有情况下都不实用,但想想还是很有趣的。



# Chapter 6: Let's Talk About Forks and Cows

这是文章的最后一部分,主要解释了进程的创建和执行过程。以下是主要内容的中文翻译,保留了原有的 Markdown 语法:

最后一个问题:我们是如何到达这里的?第一个进程从何而来?

这篇文章快要结束了。我们正在最后冲刺。即将触地得分。走向更青翠的牧场。以及各种糟糕的习语,意味着你只差第6章的长度就可以触摸草地或者做任何你不读15,000字的 CPU 架构文章时会做的事情。

如果 `execve` 通过替换当前进程来启动一个新程序,那么如何在新进程中单独启动一个新程序呢?如果你想电脑上做多件事,这是一个相当重要的能力;当你双击一个应用程序来启动它时,该应用程序会单独打开,而你之前正在运行的程序继续运行。

答案是另一个系统调用:`fork`,这是所有多进程处理的基础系统调用。`fork` 实际上很简单 - 它克隆当前进程及其内存,将保存的指令指针保留在原来的位置,然后允许两个进程照常进行。如果不加干预,这些程序会继续独立运行,所有计算都会翻倍。

新运行的进程被称为”子进程”,而最初调用 `fork` 的进程被称为”父进程”。进程可以多次调用 `fork`,因此可以有多个子进程。每个子进程都有一个进程 ID(PID),从 1 开始。

无知地复制相同的代码是相当无用的,所以 `fork` 在父进程和子进程中返回不同的值。在父进程中,它返回新子进程的 PID,而在子进程中它返回 0。这使得在新进程上做不同的工作成为可能,从而使分叉真正有用。



```

main.c

pid_t pid = fork();

// 代码从这里照常继续,但现在跨越两个"相同"的进程。
//
// 相同...除了从 fork 返回的 PID!
//
// 这是两个程序唯一的迹象表明它们不是独一无二的。

if (pid == 0) {
    // 我们在子进程中。
    // 做一些计算并将结果反馈给父进程!
} else {
    // 我们在父进程中。
    // 可能继续我们之前在做的事情。
}

```

进程分叉可能有点难以理解。从现在开始,我假设你已经弄明白了;如果你还没有,看看[这个丑陋的网站](#)获得一个相当不错的解释。

总之,Unix 程序通过调用 `fork` 然后立即在子进程中运行 `execve` 来启动新程序。这被称为 *fork-exec* 模式。当你运行一个程序时,你的计算机执行类似以下的代码:

```

launcher.c

pid_t pid = fork();

if (pid == 0) {
    // 立即用新程序替换子进程。
    execve(...);
}

// 既然我们到了这里,进程就没有被替换。我们在父进程中!
// 有用的是,我们现在在 PID 变量中有了新子进程的 PID,
// 如果我们需要杀死它的话。

// 父程序在这里继续...

```

## 哗哗!

你可能已经注意到,复制一个进程的内存只是为了在加载不同程序时立即丢弃所有内容,这听起来有点低效。幸运的是,我们有一个 MMU。复制物理内存中的数据是慢的部分,而不是复制页表,所以我们根本不复制任何 RAM:我们为新进程创建旧进程页表的副本,并保持映射指向相同的底层物理内存。

但子进程应该独立于父进程并与之隔离!子进程写入父进程的内存,或反之,都是不可以的!

引入 *COW*(写时复制)页面。使用 *COW* 页面,只要它们不试图写入内存,两个进程就从相同的物理地址读取。一旦其中一个试图写入内存,那个页面就会在 RAM 中被复制。*COW* 页面允许两个进程拥有内存隔离,而无需预先付出克隆整个内存空间的成本。这就是为什么 *fork-exec* 模式是高效的;因为在加载新的二进制文件之前没有写入旧进程的内存,所以不需要进行内存复制。

*COW* 的实现,像许多有趣的东西一样,是通过分页技巧和硬件中断处理。*fork* 克隆父进程后,它将两个进程的所有页面都标记为只读。当程序写入内存时,写入失败,因为内存是只读的。这触发了一个段错误(硬件中断类型),由内核处理。内核复制内存,更新页面以允许写入,并从中断返回以重新尝试写入。

A: 敲门!

B: 谁在那里?

A: 打断的奶牛。

B: 打断的奶牛是—

A: 哗哗!

## 在开始的时候(不是创世纪 1:1)

你电脑上的每个进程都是由一个父程序 *fork-exec* 的,除了一个:*init* 进程。*init* 进程是由内核直接手动设置的。它是第一个运行的用户空间程序,也是关机时最后被杀死的程序。

想看一个酷炫的即时黑屏吗?如果你使用的是 macOS 或 Linux,保存你的工作,打开一个终端,然后杀死 *init* 进程(PID 1):

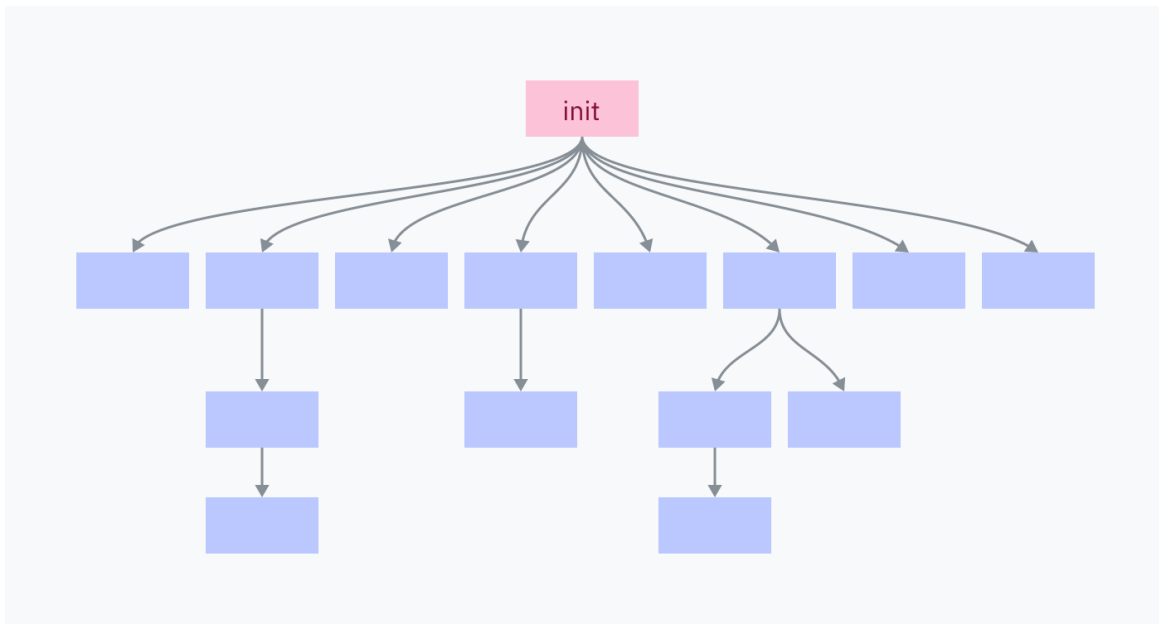
```
Shell session
```

```
$ sudo kill 1
```

作者注:关于 *init* 进程的知识,不幸的是,只适用于像 *macOS* 和 *Linux* 这样的类 *Unix* 系统。从现在开始你学到的大部分内容将不适用于理解 *Windows*,后者有一个非常不同的内核架构。

就像关于 *execve* 的部分一样,我明确提到这一点 - 我可以写另一篇关于 *NT* 内核的文章,但我正在克制自己不这样做。(暂时。)

*init* 进程负责生成构成你的操作系统的所有程序和服务。其中许多反过来又生成它们自己的服务和程序。



杀死 *init* 进程会杀死它的所有子进程及其所有子进程,从而关闭你的操作系统环境。

## 回到内核

我们[在第3章]中看 *Linux* 内核代码时玩得很开心,所以我们要再多看一些!这次我们将从内核如何启动 *init* 进程开始。

你的电脑按以下顺序启动:

1. 主板捆绑了一个微小的软件,它在你连接的磁盘中搜索一个叫做 *引导加载程序* 的程序。它选择一个引导加载程序,将其机器代码加载到 *RAM* 中,并执行它。

请记住,我们还没有进入运行中的操作系统世界。在操作系统内核启动 init 进程之前,多进程和系统调用实际上并不存在。在 pre-init 上下文中,“执行”一个程序意味着直接跳转到 RAM 中的机器代码,而不期望返回。

2. 引导加载程序负责找到一个内核,将其加载到 RAM 中,并执行它。一些引导加载程序,如 [GRUB](#),是可配置的和/或让你在多个操作系统之间选择。BootX 和 Windows Boot Manager 分别是 macOS 和 Windows 的内置引导加载程序。
3. 内核现在正在运行,开始一系列大型初始化任务,包括设置中断处理程序、加载驱动程序和创建初始内存映射。最后,内核将特权级别切换到用户模式并启动 init 程序。
4. 我们终于进入了操作系统的用户空间!init 程序开始运行 init 脚本,启动服务,并执行像 shell/UI 这样的程序。

## 初始化 Linux

在 Linux 上,步骤 3(内核初始化)的大部分工作发生在 [init/main.c](#) 中的 `start_kernel` 函数中。这个函数超过 200 行,调用了各种其他初始化函数,所以我不会在这篇文章中包含[整个内容](#),但我确实建议你浏览一下!在 `start_kernel` 的末尾,调用了名为 `arch_call_rest_init` 的函数:

```
start_kernel @ init/main.c

1087         /* Do the rest non-__init'ed, we're now alive */
1088         arch_call_rest_init();
```

## non-\_\_init'ed 是什么意思?

`start_kernel` 函数被定义为 `asmlinkage __visible void __init __no_sanitize_address start_kernel(void)`。像 `__visible`、`__init` 和 `__no_sanitize_address` 这样的奇怪关键字都是 Linux 内核中使用的 C 预处理器宏,用于向函数添加各种代码或行为。

在这种情况下,`__init` 是一个宏,指示内核在引导过程完成后立即从内存中释放该函数及其数据,仅仅是为了节省空间。

它是如何工作的?不深入细节,Linux 内核本身就是作为 ELF 文件打包的。`__init` 宏扩展为 `__section(".init.text")`,这是一个编译器指令,将代码放在一个名为 `.init.text` 的部分,而不是通常的 `.text` 部分。其他宏也允许将数据和常量放在特殊的初始化部分,例如 `__initdata` 扩展为 `__section(".init.data")`。

`arch_call_rest_init` 只是一个包装函数:

```
init/main.c

832 void __init __weak arch_call_rest_init(void)
833 {
834     rest_init();
835 }
```

注释说“do the rest non-\_\_init'ed”是因为 `rest_init` 没有用 `__init` 宏定义。这意味着它在清理 `init` 内存时不会被释放:

```
init/main.c

689 noinline void __ref rest_init(void)
690 {
```

`rest_init` 现在为 `init` 进程创建一个线程:

rest\_init @ init/main.c

```
695      /*
696      * We need to spawn init first so that it obtains pid 1, however
697      * the init task will end up wanting to create kthreads, which, if
698      * we schedule it before we create kthreadd, will OOPS.
699      */
700      pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
```

传递给 `user_mode_thread` 的 `kernel_init` 参数是一个函数,它完成一些初始化任务,然后搜索一个有效的 `init` 程序来执行它。这个过程从一些基本的设置任务开始;我大部分会跳过这些,除了调用 `free_initmem` 的地方。这就是内核释放我们的 `.init` 部分的地方!

kernel\_init @ init/main.c

```
1471      free_initmem();
```

现在内核可以找到一个合适的 `init` 程序来运行:

kernel\_init @ init/main.c

```
1495      /*
1496      * 我们尝试这些直到其中一个成功。
1497      *
1498      * 如果我们试图恢复一个真正坏掉的机器,可以使用 Bourne shell 代替 init。
1499      */
1500      if (execute_command) {
1501          ret = run_init_process(execute_command);
1502          if (!ret)
1503              return 0;
1504          panic("Requested init %s failed (error %d).",
1505                execute_command, ret);
1506      }
1507
1508      if (CONFIG_DEFAULT_INIT[0] != '\0') {
1509          ret = run_init_process(CONFIG_DEFAULT_INIT);
1510          if (ret)
1511              pr_err("Default init %s failed (error %d)\n",
1512                    CONFIG_DEFAULT_INIT, ret);
1513          else
1514              return 0;
1515      }
1516
1517      if (!try_to_run_init_process("/sbin/init") ||
1518          !try_to_run_init_process("/etc/init") ||
1519          !try_to_run_init_process("/bin/init") ||
1520          !try_to_run_init_process("/bin/sh"))
1521          return 0;
1522
1523      panic("No working init found. Try passing init= option to kernel. "
1524            "See Linux Documentation/admin-guide/init.rst for guidance.");
```

在 Linux 上,init 程序几乎总是位于或符号链接到 `/sbin/init`。常见的 init 包括 [systemd](#) (它有一个异常好的网站)、[OpenRC](#) 和 [runit](#)。如果 `kernel_init` 找不到其他任何东西,它会默认使用 `/bin/sh` - 如果它找不到 `/bin/sh`,那就真的出大问题了。

*macOS 也有一个 init 程序!它叫做 `launchd`,位于 `/sbin/launchd`。试着在终端中运行它,看看它会不会因为你不是内核而对你大喊大叫。*

从这一点开始,我们就到了启动过程的第 4 步:init 进程在用户空间中运行,并开始使用 fork-exec 模式启动各种程序。

## Fork 内存映射

我很好奇 Linux 内核在分叉进程时如何重新映射内存的下半部分,所以我稍微研究了一下。

[kernel/fork.c](#) 似乎包含了大部分分叉进程的代码。该文件的开头很有帮助地指出了正确的查看位置:

```
kernel/fork.c

8  /*
9   * 'fork.c' 包含 'fork' 系统调用的辅助例程
10  * (另见 entry.S 和其他)。
11  * 一旦你掌握了窍门,Fork 就相当简单,但内存
12  * 管理可能是个麻烦。见 'mm/memory.c': 'copy_page_range()'
13  */
```

看起来这个 `copy_page_range` 函数接收一些关于内存映射的信息并复制页表。快速浏览它调用的函数,这也是将页面设置为只读以使其成为 COW 页面的地方。它通过调用一个名为 `is_cow_mapping` 的函数来检查是否应该这样做。

`is_cow_mapping` 定义在 [include/linux/mm.h](#) 中,如果内存映射有表示内存是可写的并且不在进程间共享的**标志**,它就返回 true。共享内存不需要 COW,因为它本来就设计为共享。欣赏一下这个略微难以理解的位掩码:

```
include/linux/mm.h

1541 static inline bool is_cow_mapping(vm_flags_t flags)
1542 {
1543     return (flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
1544 }
```

回到 [kernel/fork.c](#),简单地搜索 `copy_page_range` 发现它在 `dup_mmap` 函数中被调用...而 `dup_mmap` 又被 `dup_mm` 调用...而 `dup_mm` 又被 `copy_mm` 调用...最后被庞大的 `copy_process` 函数调用!`copy_process` 是 fork 函数的核心,在某种程度上,它是 Unix 系统如何执行程序的中心点 - 总是复制和编辑在启动时为第一个进程创建的模板。



cows & cows & cows



## 总结...

### *所以...程序是如何运行的?*

在最低层面:处理器很笨。它们有一个指向内存的指针,并按顺序执行指令,除非它们遇到一条告诉它们跳到其他地方的指令。

除了跳转指令,硬件和软件中断也可以通过跳转到一个预设位置来打破执行序列,然后该位置可以选择跳转到哪里。处理器核心不能同时运行多个程序,但可以通过使用定时器重复触发中断并允许内核代码在不同的代码指针之间切换来模拟这种情况。

程序被**欺骗**相信它们作为一个连贯、隔离的单元运行。在用户模式下阻止直接访问系统资源,使用分页隔离内存空间,系统调用的设计允许通用 I/O 访问而不需要太多关于真实执行上下文的知识。系统调用是指示 CPU 运行一些内核代码的指令,其位置由内核在启动时配置。

### *但是...程序是如何运行的?*

计算机启动后,内核启动 init 进程。这是在更高抽象级别运行的第一个程序,它的机器代码不必担心许多具体的系统细节。init 程序启动渲染你计算机图形环境的程序,并负责启动其他软件。

要启动一个程序,它用 fork 系统调用克隆自己。这种克隆是高效的,因为所有内存页都是 COW 的,不需要在物理 RAM 中复制内存。在 Linux 上,这就是 `copy_process` 函数在起作用。

两个进程都检查它们是否是被分叉的进程。如果是,它们使用 `exec` 系统调用要求内核用新程序替换当前进程。

新程序可能是一个 ELF 文件,内核解析它以找到关于如何加载程序以及在新的虚拟内存映射中放置其代码和数据的信息。如果程序是动态链接的,内核可能还会准备一个 ELF 解释器。

然后内核可以加载程序的虚拟内存映射并返回用户空间,程序开始运行,这实际上意味着将 CPU 的指令指针设置到新程序在虚拟内存中的代码开始处。

# Chapter 7: Epilogue

恭喜!我们现在已经牢固地将”你”放在了 CPU 中。希望你玩得开心。

我再次强调,你刚刚获得的所有知识都是真实且活跃的。下次当你思考你的电脑如何运行多个应用程序时,我希望你能想象定时器芯片和硬件中断。当你用一些花哨的编程语言编写程序并遇到链接器错误时,我希望你能想到那个链接器正在尝试做什么。

如果你对本文中的任何内容有任何问题(或更正),你应该发邮件给我 [lexi@hackclub.com](mailto:lexi@hackclub.com) 或在 [GitHub](#) 上提交 issue 或 PR。



...但等等,还有更多!

## 附加内容: 翻译 C 概念

如果你自己做过一些低级编程,你可能知道栈和堆是什么,你可能也使用过 `malloc`。你可能没有太多思考它们是如何实现的!

首先,线程的栈是一个固定大小的内存区域,它被映射到虚拟内存的高地址处。在大多数架构中(虽然不是全部),栈指针从栈内存的顶部开始,随着地址值的增加而向下移动。值得注意的是,物理内存并不是一开始就为整个栈空间分配的。相反,系统使用按需分页技术,在实际访问到栈的某个区域时才懒惰地分配相应的物理内存。

听到像 `malloc` 这样的堆分配函数不是系统调用可能会让人惊讶。相反,堆内存管理是由 `libc` 实现提供的!`malloc`、`free` 等是复杂的程序,`libc` 自己跟踪内存映射细节。在底层,用户空间堆分配器使用包括 `mmap`(它可以映射不仅仅是文件)和 `sbrk` 在内的系统调用。

## 补充内容: 趣闻

我找不到一个连贯的地方放这些,但觉得它们很有趣,所以给你们:

大多数 *Linux* 用户可能有足够有趣的生活,他们很少花时间想象页表在内核中是如何表示的。

[\*Jonathan Corbet, LWN\*](#)

硬件中断的另一种可视化:



注意,一些系统调用使用称为 `vDSOs` 的技术,而不是跳入内核空间。我没有时间谈论这个,但它相当有趣,我建议[阅读 相关内容](#)。

最后,关于 `Unix` 的指控:我确实感到遗憾,很多特定于执行的东西都非常特定于 `Unix`。如果你是 `macOS` 或 `Linux` 用户,这没问题,但它不会让你更接近 `Windows` 如何执行程序或处理系统调用,尽管 `CPU` 架构的东西都是一样的。将来我很想写一篇涵盖 `Windows` 世界的文章。

## 致谢

在写这篇文章时,我与 GPT-3.5 和 GPT-4 进行了相当多的交谈。虽然它们经常对我撒谎,大部分信息都没用,但它们有时在解决问题时非常有帮助。如果你意识到它们的局限性并对它们说的每件事都极度怀疑,LLM 辅助可以是净正面的。话虽如此,它们在写作方面很糟糕。不要让它们为你写作。

更重要的是,感谢所有为我校对、鼓励我并帮助我头脑风暴的人 - 特别是 Ani、B、Ben、Caleb、Kara、polypixeldev、Pradyun、Spencer、Nicky(他在[第4章]画了那个精彩的精灵),以及我可爱的父母。

如果你是一个喜欢电脑的青少年,而你还没有加入 [Hack Club Slack](#),你现在就应该加入。如果我没有一个可以分享我的想法和进展的awesome社区,我就不会写这篇文章。如果你不是青少年,你应该[给我们](#)钱,这样我们就可以继续做酷的事情。

本文中所有平庸的艺术作品都是在 [Figma](#) 中绘制的。我使用 [Obsidian](#) 进行编辑,有时使用 [Vale](#) 进行 linting。这篇文章的 Markdown 源代码[在 GitHub 上可用](#)并开放给未来的吹毛求疵者,所有艺术作品都发布在 [Figma 社区页面](#)上。

