<http://cxd2014.github.io/2015/11/15/Linux-Porting/>

[Porting Linux to a new processor architecture, part 1: The basics](https://lwn.net/Articles/654783/)   
[Porting Linux to a new processor architecture, part 2: The early code](https://lwn.net/Articles/656286/)   
[Porting Linux to a new processor architecture, part 3: To the finish line](https://lwn.net/Articles/657939/)

## 移植Linux到一个新的处理器架构上

### 正文

虽然一个**简单的移植**只需要4000行左右的代码，但是让Linux内核在一个新的处理器架构上运行是一个非常困难的事情。更糟糕的是没有多少有用的文章来描述移植过程。这一系列的三篇文章的目标是提供一个移植Linux内核的大致过程，至少当你移植Linux内核到一个新的处理器架构上时可以作为一个参考。

**在花了无数的时间越来越顺畅的移植内核支持许多架构后我发现一个定义良好的框架在移植过程中可以大量参考。**这样的框架在逻辑上被分为两个紧密相连的部分。

第一部分是启动代码，架构特定的代码是在内核接管bootloader后执行直到init的最终执行。

第二部分关注的是当启动阶段已经完成内核进入正常运行状态时经常执行的架构特定代码，第二部分包含开始运行新的线程、处理硬件中断或者软件异常、复制数据给用户程序或者从用户程序复制数据、处理系统调用等等。

### 有必要进行一个全新的移植吗？

就像LWN去年发布的另一篇有关于移植的文章，移植分三种不同的层次：

移植到一个新的开发版上，它的处理器已经被支持。

移植到一个新的处理器上，但是和他架构相同的处理器系列已经被支持。

移植到一个全新的处理器架构上。

### 了解你的硬件

真正的了解底层硬件是最重要的基础，它是移植Linux最主要的前提条件

通常处理器在逻辑上或者物理上至少分为两部分。第一部分通常是用户模式下的ISA细节，这基本上意味着用户模式下处理器需要理解和执行的指令序列。第二部分描述了特权架构，他包括只能在内核模式下执行的指令序列和各个控制处理器状态的寄存器。

第二部分是移植主要关注的信息，这也是阻止开发者重用其他架构代码的主要原因。

我们需要知道的几个最主要的问题是：

1. **该处理器架构的虚拟内存模型是什么，页表的格式和翻译机制？**

许多处理器架构（例如：x86，ARM）定义了一个灵活的虚拟内存布局，他们的虚拟地址空间理论上可以任意划分为用户空间和内核空间，但是在32位处理器上Linux的默认划分方式是3GB的低地址部分划分给用户空间剩下的1GB的高端地址给Linux内核空间。在其他架构上，内存布局非常受限于硬件设计。例如MIPS32它的虚拟地址被固定分为两段相同大小的空间：低2GB空间被指定为用户空间，高2GB用于内核空间。**内核空间的物理地址甚至已经被预定义分为几个不同地址段。**

**页表的格式和处理器使用的虚拟地址到物理地址的翻译机制有紧密联系。**当使用硬件管理机制时，当TLB（一个硬件缓存用于存放最近使用的虚拟地址到物理地址的翻译）不包含一个给定的虚拟地址的翻译（称为TLB miss），硬件状态机会自动从内存中的页表结构中获取正确的翻译并将此翻译缓存到TLB中。这就意味着页表格式是固定的并且是由处理器规范定义的。当使用软件管理机制时，一个TLB miss异常是被一段代码处理的，**理论上页表的格式是怎样组织的是非常自由的，只是TLB的格式是固定的。**

1. **怎样使能或者失能中断，特权模式和用户模式之间怎样切换，怎样捕获异常等？**

虽然所有的这些操作通常只涉及在一组寄存器上读或者修改某些比特位，但是他们总是架构特定的。正是由于这个原因，大多数情况下他们是由一段专用的汇编代码实现。

1. **什么是ABI？**

虽然可能有人认为Application Binary Interface（ABI-应用程序二进制接口）的支持是编译工具链的事情，因为它定义了堆栈初始化为栈帧的方法，函数之间参数和返回值的传递方式等等。但是移植Linux内核这是完全有必要知道的。例如，作为系统调用的接受者（这通常是ABI定义的）内核需要知道在哪里得到参数，怎样返回值；或者在上下文切换时，内核必须知道哪些数据需要保存和恢复，以及线程的上下文是由什么构成的，等等。

### 了解内核

学习一点内核概念，特别是关于Linux内存布局的知识会有很大帮助。我承认我花了一段时间来弄清楚low memory和high memory，以及direct mapping（直接映射）和vmalloc regions（vmalloc区域）的区别。

一个普通的，简单的移植（在32位处理器上）内核占据虚拟地址的高端1GB地址空间，这是非常简单的。这1GB空间在Linux中定义为直接映射到物理内存的底端（称为low memory），这意味着如果内核访问地址0xC0000000，它会被重定向在物理地址的0x00000000处。

相反，在一个物理内存多余这个直接映射的区域的系统上，上面的内存区域（称为 high memory）内核不能正常访问。所以其他机制必须被使用，例如kmap()和kmap\_atomic()，用于访问这些高端内存页。

在直接映射区域之上是vmalloc区域他被vmalloc()函数控制。这种分配机制可以分配虚拟地址上的连续空间但是物理地址上是不连续的。这在要求分配大量连续的内存页但是在物理空间上没有这么多的连续空间是非常有益的。

阅读更多关于Linux内存管理的知识可以在Linux Device Drivers-PDF和这篇文章中找到。

<https://static.lwn.net/images/pdf/LDD3/ch15.pdf>

<https://lwn.net/Articles/356378/>

### 怎样开始？

当你满脑子都是处理器规定和内核规则时，该为新创建的架构目录添加一些文件了。但是等等…我应该在哪，怎样开始？所有移植甚至是所有代码都必须遵守某些API。这要分为两个步骤。

首先，一些文件和定义一些符号（函数，变量，定义）对于内核甚至编译是非常必要的。这些文件和符号可以从编译失败的信息中推到出来：**如果编译失败是因为缺少某些文件或者符号。这是一个很好的指示，你应该使用它（或者有时候一些配置选项需要改变）。**在移植Linux时这种方法是非常有效的当需要实现大量的头文件来定义架构特定的代码和内核之间的API。

编译之后的内核是能够在目标硬件上运行的，我们需要知道启动代码是非常有顺序，他允许很多函数开始时为空函数然后逐渐被实现直到系统最后变得稳定并且运行init进程。**在早期汇编启动代码执行后，运行C代码是一个常用的做法。然而early\_printk()等一些基础函数建议尽早实现不然非常难以调试。**

### 最后准备开始：最简易的一组无代码文件

移植编译工具到一个新的处理器架构上是移植Linux内核的前提条件，这里我们假设已经完成。就编译工具而言最后需要做的是建立交叉编译器。**移植C标准库还没有完成，仅仅完成了交叉编译器的阶段一。**

**这样的交叉编译器只能编译裸机代码，但是非常适合编译内核因为内核不需要依赖任何外部库。相反，处于阶段二的交叉编译器是用来支持C标准库的。**

移植Linux到一个新的处理器架构的第一个步骤是在内核源码树根目录下的arch/目录下新建一个目录（例如我建立的是linux/arch/tsar/），在这个新目录中它的文件布局是非常标准的：

configs/: 支持Linux系统的默认配置（即 \*\_defconfig 文件）

include/asm/： 仅供内部使用的头文件，即 Linux源文件

include/uapi/asm： 需要提供给用户空间的头文件（例如 libc库）

kernel/: 通用内核管理

lib/: 架构特定的优化程序（例如 memcpy(), memset()）

mm/: 内存管理

一旦新的架构目录出现Linux自动知道它的存在。它只是抱怨没有找到新架构下的Makefile文件：

~/linux $ make ARCH=tsar

Makefile: ~/linux/arch/tsar/Makefile: No such file or directory

下面的例子是一个最简化的Makefile文件只有几个变量：

KBUILD\_DEFCONFIG := tsar\_defconfig

KBUILD\_CFLAGS += -pipe -D\_\_linux\_\_ -G 0 -msoft-float

KBUILD\_AFLAGS += $(KBUILD\_CFLAGS)

head-y := arch/tsar/kernel/head.o

core-y += arch/tsar/kernel/

core-y += arch/tsar/mm/

LIBGCC := $(shell $(CC) $(KBUILD\_CFLAGS) -print-libgcc-file-name)

libs-y += $(LIBGCC)

libs-y += arch/tsar/lib/

drivers-y += arch/tsar/drivers/

1. KBUILD\_DEFCONFIG必须是一个有效的默认配置文件名，就是configs目录下的默认配置文件(configs/tsar\_defconfig)
2. KBUILD\_CFLAGS和KBUILD\_AFLAGS 定义编译选项，分别对应编译器和汇编器。
3. {head,core,libs,...}-y 列出的是被编译进内核映象的目标文件（或者是子目录名）详细请看Documentation/kbuild/makefiles.txt文件

arch/目录下的文件Kconfig有两个用途：架构特定的配置选项的帮助文档；选择架构无关的配置选项（即 那些在其他Linux源码中早已定义的选项）适用于这个架构。

由于它是新创建架构的主要配置文件，他的内容决定了menuconfig命令的布局（例如 make ARCH=tsar menuconfig）。在这个文件中怎加内容是非常困难的因为他非常依赖于特定架构，但是看这个文件中用于其他架构的选项是非常有帮助的。

defconfig文件（例如 configs/tsar\_defconfig）对于Linux内核编译系统（kbuild）是有重要的。他的作用是定义这个架构的默认配置选项，它用于以这个基本配置为种子来配置生成一个全配置的Linux内核。可以参考其他架构下的defconfig文件。但是我们需要提炼他，因为对于支持例如：USB，IOMMU甚至文件系统对于这个阶段来说太早了。

最后“不是真正的代码但是非常重要”的是创建一个脚本文件（通常放在kernel/vmlinux.lds.S）他会指导连接器怎样放置代码和数据的各个段在最后的内核映象中。例如，汇编启动代码必须放在二进制文件的最开始位置，是这个文件允许我们这样做。

总结

到达这一步后，编译系统已经可以使用了。现在可以生成一个初步的内核配置文件，定制它甚至可以通过它来编译内核。但是编译器会很快停止因为到目前为止还么有包含任何代码。

下一篇文章介绍移植的第二阶段，我会添加一些代码包括：头文件、早期汇编启动代码和所有重要的会被执行的函数直到第一个内核线程被创建。

## 移植Linux到一个新的处理器架构上

### 正文

在part 1中我们通过说明（代码无关）的初步步骤为移植Linux到一个新的处理器架构建立了一个基础。这篇文章接着上一篇开始研究启动代码，他包括从汇编启动代码到创建第一个内核线程我们需要怎样写代码。

### 头文件

就像上一篇文章提到的，arch目录下的头文件（我把它们放在linux/arch/tsar/include/）由Linux内核要求的架构特定和架构无关代码之间的接口组成。

**第一部分头文件（子目录asm/）是内核接口的一部分它被Linux内核内部使用。第二部分(uapi/asm/)是用户接口的一部分，这意味着这些头文件会暴露在用户空间下，尽管各种C标准库试图重新实现这些头文件而不是包含这些头文件。这些接口不是完全封闭的，很多asm目录下的头文件可以被用户空间使用。**

两个接口的头文件数量总共超过100个，也就是为什么头文件是移植Linux的最大任务之一。幸运的是近几年开发者注意到很多处理器架构可以相互分享相似的代码（因为他们经常表现相同的行为），所以这些代码被汇总到通用头文件层（linux/include/asm-generic/和linux/include/uapi/asm-generic/）

真正有益的是我们可以参考这些通用头文件，而不是通过写适当的Kbuild文件来提供制定版本。例如，include/asm/Kbuild典型的头几行代码像这样：

generic-y += atomic.h

generic-y += barrier.h

generic-y += bitops.h

...

当移植Linux时我担心我只能通过列出所有可能的头文件然后一个一个的检查他们是否可以被通用版本使用还是需要自定义。**这样的列表可以根据Linux早已提供的通用头文件创建以及那些自定义头文件可以参考其他架构。**

基本上一个特定版本的开发必须要有所有关于这个架构细节的头文件，缓存cache(asm/cache.h)和TLB管理(asm/tlbflush.h),ELF格式(asm/elf.h),使能/失能中断(asm/irqflags.h),页表管理(asm/page.h, asm/pgalloc.h, asm/pgtable.h),上下文切换(asm/mmu\_context.h, asm/ptrace.h),字节顺序(uapi/asm/byteorder.h),等等.

### 启动顺序

正如part 1说明的，弄清楚启动顺序对于理解必须按顺序实现的最小架构特定函数集是非常有帮助的。

启动顺列的第一个函数必须亲自写出来通常用汇编代码（我写的函数名叫kernel\_entry()放在arch/tsar/kernel/head.S文件中）。它被定义为当bootloader将内核映象加载到内存后跳转到内核映象的入口点。

下面列出了一系列在启动时需要被执行的函数（被标记的函数是架构特定的函数稍后会继续讨论）：

kernel\_entry\*

start\_kernel

setup\_arch\*

trap\_init\*

mm\_init

mem\_init\*

init\_IRQ\*

time\_init\*

rest\_init

kernel\_thread

kernel\_thread

cpu\_startup\_entry

### 早期汇编启动代码

汇编启动代码这个特殊的光环使我刚开始对它产生了恐惧（我确信很多其他的程序员也有是这样），因为它经常被认为是移植过程中最复杂的代码之一。虽然写汇编代码不是一件容易的事情但是早期启动代码不是魔术。它仅仅是执行第一个架构特定的C函数的跳板，因此只需要执行一个短的定义好的任务列表。

当早期启动代码开始执行时，它不知道之前发生了什么事：系统是重启还是刚刚开机？是哪个bootloader将内核加载到内存？等等，由于这个原因将处理器设置为已知的状态是安全的。重新设置一个或者几个系统寄存器就可以达到目的，确保处理器处于内核模式并且中断是关闭的。

**相似的它也不知道内存的状态。尤其是没有保证放置内核bss段的内存处是否初始化为零，这就是为什么这个段必须清零.**

通常Linux接受bootloader传递的参数（和程序启动时接受参数的方法是一样的）。例如，**这可能是一个flattened device tree（FDT）的内存地址（ARM，MicroBlaze，openRISC等等）或者是一些其他的架构特定的结构体。通常这样的参数是通过寄存器传递然后保存到适当的内核变量中。**

此时虚拟内存还没有被激活，有趣的是注意观察内核符号他们都被定义在内核虚拟地址空间中，必须通过一个特殊的宏来访问它：x86是pa(), OpenRISC是tophys()等等。这个宏将内核符号的虚拟地址翻译为对应的物理地址，它作为一个临时的基于软件的翻译机制。

**为了使能虚拟内存，页表结构体必须从头开始设置。这个结构体通常作为一个静态变量存放在内核映象中，因为在这个阶段几乎不可能分配内存。相同的原因只有内核映象可以首先通过页表映射，尽可能使用大页内存。根据惯例这个初始页表结构体被swapper\_pg\_dir函数调用然后在整个系统运行过程中作为参考页表结构体。**

**在许多处理器架构中一个有趣的事情是内核实际上需要被映射两次。第一次映射就是part 1描述的直接映射策略（即 访问虚拟地址0xC0000000被重定向为物理地址0x00000000）。然而另一次映射是临时的当虚拟内存刚刚被使能时但是执行代码还没有跳转到虚拟内存处。第二次映射是一个简单的象征性映射（identity mapping）（即 访问虚拟地址0x00000000被重定向为物理地址0x00000000）**

页表结构体已经初始化完成现在可以使能虚拟内存，这意味着内核现在全部运行在虚拟地址空间并且所有内核符号可以通过它的名字正常访问，不需要使用早先的宏翻译方法。

最后一步之一是设置栈寄存器的地址为初始内核栈这样就可以调用C函数了。在许多处理器架构中（SPARC, Alpha, OpenRISC, etc.），另一个寄存器专门用于保存当前线程信息的指针（thread\_info结构体）。设置这个指针是可选的，因为它可以被当前内核栈指针推导出来（thread\_info结构体通常放置在内核栈的底部），但是当这个架构允许它可以更加快速和方便的访问。

早期启动代码的最后一步是跳转到Linux提供的第一个架构无关的C函数start\_kernel()处。

### 创建内核第一个线程的过程

**start\_kernel()是很多子系统初始化的地方，各种虚拟文件系统缓存和时钟管理的安全框架，控制层等等。在这里我们主要看start\_kernel()在最后调用rest\_init()前调用架构特定的几个函数，rest\_init()函数首先创建连个内核线程然后变为idle线程（空闲线程-当CPU空闲时运行此线程）。**

#### setup\_arch()

setup\_arch()别看它的名字普通但是做了很多架构特定的事情。当你观察不同架构下的代码时你会发现他们通常做相同的事情，尽管使用不同的方法和顺序。当做一个简单的移植时可以参考setup\_arch()这个简单的框架。

第一个步骤是知道系统内存的大小。**一个基于设备树（device-tree-based）的系统可以快速浏览（使用early\_init\_devtree()）bootloader提供的tag参数列表（flattened device tree）来得到可用的物理内存块然后将他们注册到memblock层。**接下来解析（使用parse\_early\_param()）bootloader提供或者是直接包含在设备树中的可以激活有用的特性例如early\_printk()的启动参数。这里顺序是非常重要的因为设备树可能包含终端设备用于打印显示的物理地址，因此首先需要扫描一遍。

接下来memblock层在映射低端内存（low memory）区域前需要进一步配置，使内存可以被分配。首先，被内核映象和设备树占用的内存区域会被设置为保留区域以便于稍后被伙伴分配器（buddy allocator）从空闲内存池中移除。高端内存和低端内存的分界线（即 哪个物理内存区域是直接映射区）必须确定下来。最后页表结构体可以被清除（清除早期启动代码创建的identity mapping）然后映射低端内存区。

内存的最后一步初始化是配置内存区域。物理内存页和不同区域关联：ZONE\_DMA兼容老的ISA 24-bit DMA地址限制，ZONE\_NORMAL和ZONE\_HIGHMEM分别对应低端和高端内存页，更多关于Linux内存分配的知识请看Linux Device Drivers [PDF]。

最后内核内存段可以使用源码API和flattened device tree创建的结构体device\_node进行注册。

如果使能了early\_printk()，这个例子是展示终端在这个阶段的输出信息：

Linux version 3.13.0-00201-g7b7e42b-dirty (joel@joel-zenbook)

(gcc version 4.8.3 (GCC) ) #329 SMP Thu Sep 25 14:17:56 CEST 2014

Model: UPMC/LIP6/SoC - Tsar

bootconsole [early\_tty\_cons0] enabled

Built 1 zonelists in Zone order, mobility grouping on. Total pages: 65024

Kernel command line: console=tty0 console=ttyVTTY0 earlyprintk

#### trap\_init()

trap\_init()的作用是配置中断/异常相关的硬件和软件架构特定的部分。此时一个异常要么使系统立即崩溃或者被bootloader设置的处理函数捕获（最终还是导致系统崩溃但是可能提供更多的信息）。

Linux移植过程中在trap\_init()后面隐藏了另一段更加复杂的代码：中断/异常管理器。它的一大部分必须使用汇编代码编写因为就像早起启动代码一样它处理的是目标架构处理器的特定部分。一个典型的处理器，当中断时会发生以下事情：

1. 处理器自动切换到内核模式，禁止中断然后跳转到特定地址处加载中断处理程序。
2. 主中断处理函数检查是哪个中断发生了然后跳转到相应的子处理函数。中断向量表经常用来关联特定的处理函数，所以在一些架构上没有主中断处理函数，实际中断事件和中断向量表之间是由硬件自动完成对应关系。
3. 子中断处理函数保存当前上下文，处理器状态会被保存起来以便恢复中断。也可能使能中断（使Linux可重入）然后通常跳转到C函数中更好的处理异常。例如当用户程序访问非法内存时C函数可以使用SIGBUS信号终止用户程序。

一旦所有中断基础设施到位trap\_init()初始化中断向量表然后通过一个系统寄存器配置处理器映射主中断处理程序的地址（或者直接是中断向量表的地址）。

#### mem\_init()

mem\_init()的作用是从memblock层释放空闲内存给buddy分配器（又名页分配器）。slab分配器（常用对象的缓存，通过kmalloc()访问）和vmalloc都是基于buddy分配器完成这个最后的任务开始运行的。

mem\_init()通常打印内存系统的一些信息：

Memory: 257916k/262144k available (1412k kernel code, \

4228k reserved, 267k data, 84k bss, 169k init, 0k highmem)

Virtual kernel memory layout:

vmalloc : 0xd0800000 - 0xfffff000 ( 759 MB)

lowmem : 0xc0000000 - 0xd0000000 ( 256 MB)

.init : 0xc01a5000 - 0xc01ba000 ( 84 kB)

.data : 0xc01621f8 - 0xc01a4fe0 ( 267 kB)

.text : 0xc00010c0 - 0xc01621f8 (1412 kB)

#### init\_IRQ()

中断网络可以是非常困难和复杂。在一个简单的系统中少量硬件设备的中断线直接连接到处理器的中断入口。在复杂的系统中大量硬件设备连接到众多可编程中断控制器(PICs)上，这些PICs经常相互级联，组成一个多层的中断网络。设备树（device tree）我们让可以简单的描述这个网络而不是在源码中直接指定他们。

init\_IRQ()函数主要的任务是调用irqchip\_init()函数来扫描设备树找到所有标明为中断控制器的节点(e.g PICs)。然后找到每个节点对应的驱动初始化它。除非目标系统使用早已被支持的中断控制器，通常意味着第一个设备驱动程序需要自己编写。

这样的驱动程序包含一些这样函数：一个初始化函数它将设备映射到内核地址空间也将控制器局部中断线映射到内核IRQ中断号空间（使用irq\_domain映射库）；一个mask/unmask函数它可以配置控制器屏蔽或者不屏蔽指定Linux中断号；最后还有一个控制器指定中断处理函数它可以找到哪个输入时激活的然后调用这个输入注册的中断处理函数（例如，这就是连接到PIC上的块设备触发一个中断时相应的中断处理函数怎样被调用的）

#### time\_init()

time\_init()函数的作用是初始化timekeeping基础设施的架构特定部分。这个函数的最简版本是依靠设备树仅仅调用连个函数。

首先of\_clk\_init()函数会扫描设备树然后找到所有标明为时钟源的节点然后初始化这个时钟框架**。一个非常简单的时钟源节点仅仅定义直接表明它性能的一个固定频率。**

**然后clocksource\_of\_init()会解析设备树的时钟源节点然后初始化他们相应的驱动。正如内核文档中描述的，linux实际上需要两种timekeeping抽象（他们通常由相同的设备提供）：一个时钟源设备提供monotonically counting（单调计数？）的基本时间表（例如它可以计算系统的周期），另一个时钟事件设备在时间表上的确定时刻触发一个中断，特别是通过编程设定的时间周期。通过时钟源可以允许精确计时。**

时钟源设备的驱动程序可以非常简单，特别是对于一个内存映射的设备，通用MMIO时钟源驱动程序只需要知道设备计数器的寄存器地址。对于时钟事件稍微有些复杂因为驱动程序需要定义怎样编写一个周期和怎样知道它已经结束，以及当定时器中断触发时提供一个中断处理函数。

### 总结

start\_kernel()函数的一个主要任务是校准一个jiffy的循环数，它是处理器执行内部延时一个jiffy的循环数–一个内部时钟周期通常在1到10个毫秒范围内。成功的实现了校准应该意味着不同的基础设备和驱动已经通过我们刚才提到的架构特定函数设置好了，因为校准需要用到他们。

在下一篇文章中我会展示移植的最后一部分：从创建第一个内核线程到执行init进程。

## 移植Linux到一个新的处理器架构上

### 正文

这一系列文章提供了一个移植Linux内核到新的处理器架构上的大概流程。Part 1和part 2分别介绍了代码无关的基础工作和从汇编启动代码到创建第一个内核线程的早期代码。这是最后一篇文章主要介绍启动init进程进行线程和进程的管理工作。

### 启动内核线程

当start\_kernel()函数调用了最后一个函数rest\_init()时，内存管理子系统全面运行了，处理器开始运行并且可以处理异常和中断，系统已经具备时钟概念。

但是执行流到目前为止还是单线程的，rest\_init()函数在进入idle线程之前的主要任务是创建两个内核线程：kernel\_init它会在下一节进行讨论和kthreadd。你可以想象，创建这些线程（和其他各种线程，用户线程也是通过相同的方式创建的）需要一个复杂的进程管理体系。创建一个新的线程的大部分代码是架构无关的：例如复制task\_struct结构体或者证书，设置调度器等等通常不需要架构特定的代码。然而，进程管理代码必须定义一些架构特定的部分，主要是为新线程设置栈和线程之间的切换。

Linux总是避免从头创建新资源尤其是新的线程。初始线程（这个线程正在启动系统）是个例外，内核总是复制已有线程然后改造为新线程。同样的原则应用于创建线程后当新线程第一次执行时恢复线程的执行比从头开始执行要容易。意思就是**新线程的第一次运行时新分配的栈必须先初始化，使线程看起来像被停止后在重新恢复运行。**

为了进一步理解这种机制，了解了一些线程切换机制然后再深入了解架构特定的上下文函数switch\_to()是必须的。这个函数通常是用汇编代码写的，它总是由当前线程调用然后由下一个线程返回。这个功能的部分实现是通过保存当前上下文到当前线程的堆栈中，切换堆栈指针指向下一个线程的堆栈，然后恢复被保存的上下文。 由于这是一个特殊函数，switch\_to()返回调用函数的方法是使用新的当前线程堆栈中保存的指令地址。

在这种情况下，下一个线程是先前运行过的然后被暂时移出处理器，返回调用函数是一个正常事件最终会使线程恢复自身代码的执行。然而作为一个新的线程，他还没有调用switch\_to()函数来保存线程上下文。这就是为什么新线程的栈必须初始化用来假装它以前已经调用过函数，使这个新线程恢复运行后switch\_to()可以返回。这种函数通常设置为少量汇编代码跳转到线程的代码中。

**注意内核线程切换时通常不涉及页表的切换因为是在内核地址空间中，所有内核线程的运行定义在每个页表结构体中。对于用户进程，切换他们自己的页表是通过架构特定函数switch\_mm()完成的。**

### 第一个内核线程

就像源码中解释的那样内核线程kernel\_init第一个被创建的原因是它必须获得PID 1。这是init进程的PID（即 第一个用户空间进程由kernel\_init创建）

有趣的是kernel\_init的第一个任务是等待第二个内核线程kthreadd的完成。kthreadd是内核线程的守护进程负责异步生成内核线程。一旦kthreadd开始运行，kernel\_init继续进行第二阶段的引导，他包含一点架构特定的初始化。

在多核处理器系统中，kernel\_init首先启动其他处理器核然后初始化构成驱动模型的各个子系统（devtmpfs, devices, buses, etc.）最后使用已定义的初始化调用来初始化实际底层硬件设备驱动程序。在进入设备驱动程序（e.g. block device, framebuffer, etc.）之前，至少初始化一个操作终端（通过安装相应的驱动程序）是一个好的主意。尤其是early\_printk()函数设置的早期终端应该被一个真正的全功能的终端所取代。

**也是通过这些初始化调用来解压initramfs和挂载这个初始根文件系统（rootfs）。挂载初始rootfs有几种选择但是我发现initramfs是当移植Linux时最简单的方法。这个rootfs会直接编译进内核二进制映象中。挂载之后这个rootfs可以访问/init和/dev/console。**

最后init段的内存会被释放（即 这段内存中包含的是只在初始化阶段使用以后不需要的代码和数据）然后启动在rootfs中找到的init进程。

### 运行init进程

此时启动init进程当试图取第一条指令时可能导致错误。这是因为运行init进程（实际上是所有用户空间的应用程序）首先需要涉及一点基础设施。

解决取指令问题的函数实际上就是需要处理内存页缺失异常。Linux非常懒，尤其是运行用户程序时默认情况下Linux不会预加载代码和数据到内存。它只设置所有必需的内核结构体然后让应用程序在取第一条指令时发生异常因为包含应用程序的文本段内存页通常还没有被加载。

这是实际上是故意这样设计的因为当发生内存故障时会被页故障处理函数捕获。这个处理函数可以看作是一个复杂的switch语句它可以所有内存故障：来自vmalloc()的故障会同步参考页表来扩展用户应用程序的堆。在这种情况下处理函数会确认页故障对应的应用程序的有效虚拟内存区域（VMA）然后加载缺失页到内存在次运行应用程序。

一旦页故障处理函数可以捕获内存故障，一个非常简单init进程好像可以运行。然而，它不能做很多事情因为还不能通过系统调用来请求任何服务，例如打印字符到终端。为此系统调用必须完成架构特定的部分。系统调用被视为软件中断因为他们使用用户指令使处理器自动切换到内核模式，就像硬件中断那样。除了定义支持系统调用的列表，处理系统调用还需要增加中断和异常处理函数的额外功能来接受系统调用引起的异常。

一旦支持系统调用现在应该可以运行一个"hello world"版的init程序它可以打开主控制台然后输入信息。但是仍然不能运行可以启动其它应用程序和相互通信以及和内核交换数据的全功能init程序。

实现这个目标的第一步是关注信号的管理更具体的是信号的传递（传递给另一个进程或者传递个内核自己）。如果一个进程定义了一个特定信号的处理函数那么只要给定信号还没有处理此函数会被调用。当这种事件出现在目标进程需要再次被调度时。进一步说这意味着在恢复进程时，就在返回用户模式的瞬间，为了执行处理函数这个进程的执行流必须被改变。还必须在这个应用程序的栈上增加空间来执行这个处理函数。一旦处理函数执行完成返回给内核（通过早已加入这个处理函数的上下文中的系统调用），这个进程的上下文被被加载于是它可以恢复正常运行。

运行用户空间应用程序的第二和最后一个步骤是处理用户空间内存访问：当内核想从用户空间内存页中复制数据。这样的操作可能是非常危险的，例如应用程序提供一个假指针内核如果没有正确检查会导致内核恐慌（或者是安全漏洞）。为了避免这种问题有必要编写一个架构特定函数使用一些assembly magic（汇编魔术？？）在异常表中注册所有指令执行时访问的用户空间内存地址。就像2001年的LWN文章中解释的“**如果一个故障发生在内核模式下，故障处理函数会扫描异常表试图使用一个表项匹配故障指令的地址。如果找到匹配项，一个特殊的错误会出现，这个复制操作会优雅的失败，这个系统调用会返回一个段错误”**

### 总结

一旦全功能的init进程可以运行并且访问shell，这可能是移植过程结束的信号。但是这更像是这场冒险的开始，如果这个移植需要维护（因为内部APIs有时候变化的很快），也可以通过很多方法来完善：增加支持多核处理器和NUMA系统，实现更多设备驱动程序等等。

通过描述移植Linux到一个新的处理器架构上，我希望这一系列文章有助于弥补内核文档在这方面的缺失来帮助下一个从事这个挑战的程序员，但是最终的收获是经验。

<https://lwn.net/Articles/597351/>