项目概述和开发环境配置

这是一篇阐述如何在基于 Intel x86 架构的 IBM PC 机及其兼容计算机上构建一个简单的操作系统内核的本科毕业设计论文。我将带领大家一起来探索 x86 CPU 的保护模式下操作系统内核的编写方法，一起感受一次完整的探索过程。虽然这个小内核和一个具有商业价值的操作系统内核相较而言依旧相差甚远。但是通过这样的探索，相信我们能充分的理解 x86 保护模式的运行方式和操作系统的基本原理，而这恰恰是传统的通过理论教学和阅读书籍的方式难以获得的深刻体验。

言归正传，开始我的介绍。工欲善其事，必先利其器，我先来阐述下开发环境和相关的工具配置。

工作环境

Windows 和 Linux 之争由来已久，我不想在这篇论文里针对这个问题再费口舌，我的工作环境选择 Linux。使用 Linux 的原因很简单，这里有可以自由使用的一系列的开源软件能很好的协助我们的开发和调试工作，而在 Windows 下缺乏相应的免费工具。虽然我的构建环境使用的是 Ubuntu-16.04-i386-Desktop，但是这不影响大家在对项目验证时对于 Linux 发行版的选择，因为使用的命令基本上都是相同的。经过四年本科的学习，我们对一些 Linux 基础的命令和基本的计算机概念有一定的理解和掌握，包括而不限于：

- 熟悉微机原理和基本的操作系统原理，了解基本的计算机原理概念。
- 了解和熟悉 Intel x86 保护模式下的一些名词和概念，至少需要熟悉 Intel 8086.
- 熟悉和掌握 Linux 的常用命令，能在 Linux 下进行基本的系统程序编写。
- 掌握简单的 x86 汇编语言，能读懂和编写简单的汇编程序（至少能看懂）。
- 熟练掌握 C 语言程序的编写，对 C 语言中较为复杂的语法有所了解。
- 理解和掌握 C 语言程序编译的过程，了解链接的基本原理。
- ……

学习本来就是一个从无到有的过程，操作系统内核的编写本来就是一个及其复杂和麻烦的过程。我会尽量降低这个小内核的难度，给充满热情但相关基础较为薄弱的读者阐述尽可能多的背景资料和原理解析（至少也会给出参考资料的链接）。我相信哪怕你之前的基础再弱，至少也能"照猫画虎"的构建出一个可以在裸机上运行的小内核。尽管我做的东西甚至只是一个基本原理的演示，但那也是实打实的可以运行在裸机上的小内核。

开发语言

开发工具

接着是选择开发使用的工具了，这个我简单罗列出来。首先 C 语言的编译器肯定使用 gcc，链接器自然也就是 ld 了。同时大项目自然也少不了 GNU make 这个构建工具。至于汇编编译器我们选择 nasm 这个开源免费的编译器，以便使用大多数读者习惯的 Intel 风格的汇编语法。不过考虑到需要在一些 C 语言代码中内联汇编指令，而 gcc 使用的是 AT&T 风

格的汇编语法，所以我还是稍稍学习掌握了一部分的 AT&T 风格的语法。这些就是开发使用的基本工具了，其他的工具我会在使用的时候再介绍。

我们写用户级别程序自然可以直接运行，现在是要写一个操作系统内核。我们在哪里运行它？我们可以使用虚拟机。不过我们这次使用的不是大多数人熟悉的 Vmware 或者 Virtual Box，而是一款叫做 bochs 的虚拟机。为什么呢？因为有调试的需要。我们需要一个能调试其上运行着的操作系统的虚拟机，而 bochs 是个不错的选择。选择另一款叫做 qemu 的虚拟机也支持调试，但本着简单易用的原则，这里不选择 qemu。

bochs 的安装方法很简单，以 Ubuntu 为例，只需执行以下命令即可。

```
sudo apt-get install bochs bochs-x
```

开发中用到的脚本文件

Makefile

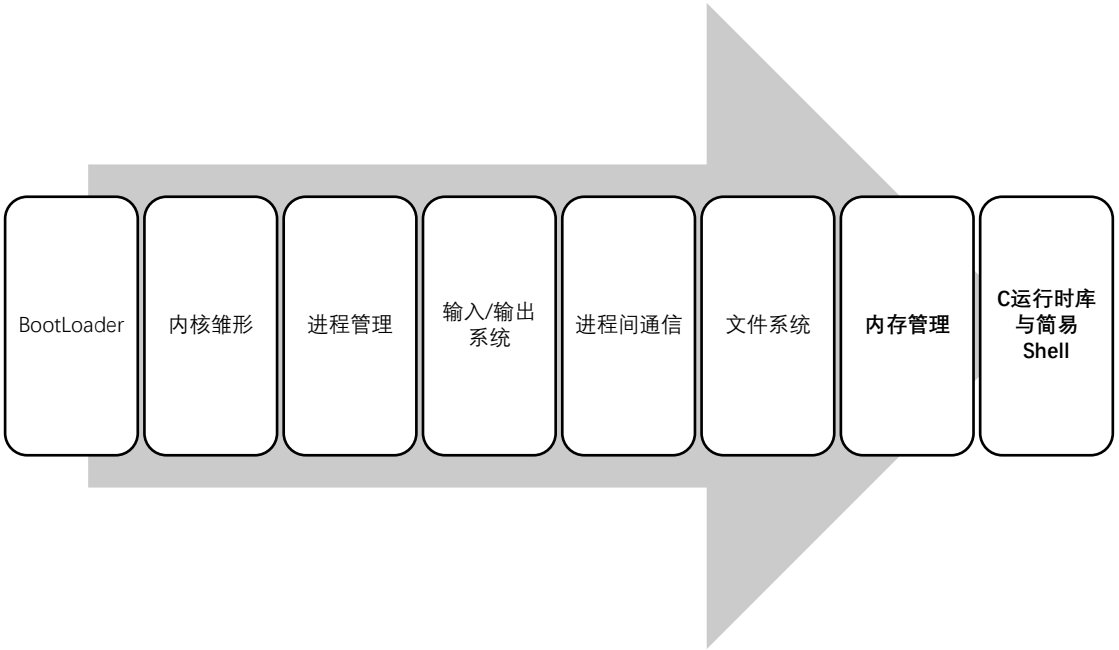
bochs 虚拟机的配置文件

总体设计

MOS 目前支持的硬件环境是基于 Intel 80386 以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现 MOS 的过程逐渐展开介绍。那我们准备如何一步一步实现 MOS 呢？按照一个操作系统的开发过程，我们可以有如下的开发步骤：

1. bootloader：理解操作系统启动前的硬件状态和要做的准备工作，了解运行操作系统的外设硬件支持，操作系统如何加载到内存中，理解两类中断--“外设中断”，“陷阱中断”，内核态和用户态的区别；
2. 内核雏形：
3. 进程管理：
4. 输入/输出系统：
5. 进程间通信：
6. 文件系统：理解文件系统的具体实现，与进程管理和内存管理等的关系，缓存对操作系统 IO 访问的性能改进，虚拟文件系统（VFS）、buffer cache 和 disk driver 之间的关系。
7. 内存管理：
8. C 运行时库与简易 Shell：

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示：



x86 处理器背景知识

这一章的内容与操作系统原理相关的部分较少，与计算机体系结构（当然也就是 x86）的细节相关的部分较多。但这些内容对写一个操作系统关系较大，要知道操作系统是直接与硬件打交道的软件，所以它需要“知道”需要硬件细节，才能更好地控制硬件。另一方面，部分内容涉及到操作系统的重要抽象--中断类异常，能够充分理解中断类异常为以后进一步了解进程切换、上下文切换等概念会很有帮助。

Intel 80386 运行模式

一般 CPU 只有一种运行模式，能够支持多个程序在各自独立的内存空间中并发执行，且有用户特权级和内核特权级的区分，让一般应用不能破坏操作系统内核和执行特权指令。80386 处理器有四种运行模式：实模式、保护模式、SMM 模式和虚拟 8086 模式。这里对涉及 MOS 的实模式、保护模式做一个简要介绍。

实模式：这是个人计算机早期的 8086 处理器采用的一种简单运行模式，当时微软的 MS-DOS 操作系统主要就是运行在 8086 的实模式下。80386 加电启动后处于实模式运行状态，在这种状态下软件可访问的物理内存空间不能超过 1MB，且无法发挥 Intel 80386 以上级别的 32 位 CPU 的 4GB 内存管理能力。实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。

对于 MOS 其实没有必要涉及，这主要是 Intel x86 的向下兼容需求导致其一直存在。其他一些 CPU，比如 ARM、MIPS 等就没有实模式，而是只有类似保护模式这样的 CPU 模式。

保护模式：保护模式的一个主要目标是确保应用程序无法对操作系统进行破坏。实际上，80386 就是通过在实模式下初始化控制寄存器（如 GDTR, LDTR, IDTR 与 TR 等管理寄存器）以及页表，然后再通过设置 CR0 寄存器使其中的保护模式使能位置位，从而进入到 80386

的保护模式。当 80386 工作在保护模式下的时候，其所有的 32 根地址线都可供寻址，物理寻址空间高达 4GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。保护模式下 80386 支持多任务，还支持优先级机制，不同的程序可以运行在不同的特权级上。特权级一共分 0 ~ 3 四个级别，操作系统运行在最高的特权级 0 上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享也可以很好地隔离各个任务。

Intel 80386 内存架构

地址是访问内存空间的索引。一般而言，内存地址有两个：一个是 CPU 通过总线访问物理内存用到的物理地址，一个是我们编写的应用程序所用到的逻辑地址（也有人称为虚拟地址）。比如如下 C 代码片段：

```
int boo=1;
int *foo=&a;
```

这里的 boo 是一个整型变量，foo 变量是一个指向 boo 地址的整型指针变量，foo 中储存的内容就是 boo 的逻辑地址。

80386 是 32 位的处理器，即可以寻址的物理内存地址空间为 2\^32=4G 字节。为更好理解面向 80386 处理器的 ucore 操作系统，需要用到三个地址空间的概念：物理地址、线性地址和逻辑地址。物理内存地址空间是处理器提交到总线上用于访问计算机系统中的内存和外设的最终地址。一个计算机系统中只有一个物理地址空间。线性地址空间是 80386 处理器通过段（Segment）机制控制下的形成的地址空间。在操作系统的管理下，每个运行的应用程序有相对独立的一个或多个内存空间段，每个段有各自的起始地址和长度属性，大小不固定，这样可让多个运行的应用程序之间相互隔离，实现对地址空间的保护。

在操作系统完成对 80386 处理器段机制的初始化和配置（主要是需要操作系统通过特定的指令和操作建立全局描述符表，完成虚拟地址与线性地址的映射关系）后，80386 处理器的段管理功能单元负责把虚拟地址转换成线性地址，在没有下面介绍的页机制启动的情况下，这个线性地址就是物理地址。

相对而言，段机制对大量应用程序分散地使用大内存的支持能力较弱。所以 Intel 公司又加入了页机制，每个页的大小是固定的（一般为 4KB），也可完成对内存单元的安全保护，隔离，且可有效支持大量应用程序分散地使用大内存的情况。

在操作系统完成对 80386 处理器页机制的初始化和配置（主要是需要操作系统通过特定的指令和操作建立页表，完成虚拟地址与线性地址的映射关系）后，应用程序看到的逻辑地址先被处理器中的段管理功能单元转换为线性地址，然后再通过 80386 处理器中的页管理功能单元把线性地址转换成物理地址。

页机制和段机制有一定程度的功能重复，但 Intel 公司为了向下兼容等目标，使得这两者一直共存。

上述三种地址的关系如下：

- 分段机制启动、分页机制未启动：逻辑地址--->*段机制处理*--->线性地址=物理地址
- 分段机制和分页机制都启动：逻辑地址--->*段机制处理*--->线性地址--->*页机制处理*--->物理地址

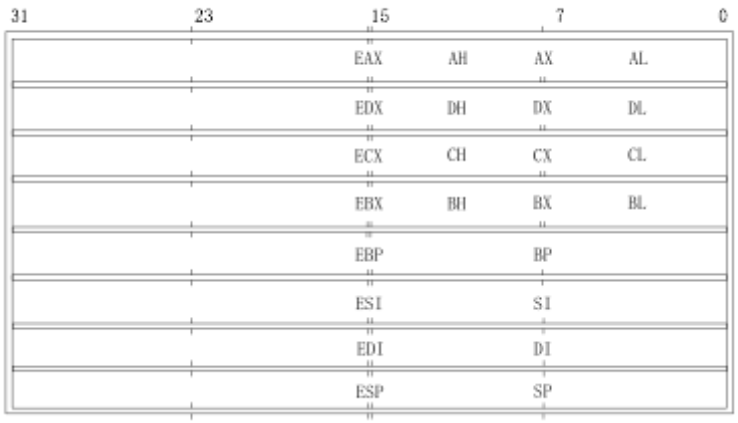
Intel 80386 寄存器

这里假定读者对 80386 CPU 有一定的了解，所以只作简单介绍。80386 的寄存器可以分为 8 组：通用寄存器，段寄存器，指令指针寄存器，标志寄存器，系统地址寄存器，控制寄存器，调试寄存器，测试寄存器，它们的宽度都是 32 位。一般程序员看到的寄存器包括通用寄存器，段寄存器，指令指针寄存器，标志寄存器。

General Register(通用寄存器)：EAX/EBX/ECX/EDX/ESI/EDI/ESP/EBP 这些寄存器的低 16 位就是 8086 的 AX/BX/CX/DX/SI/DI/SP/BP，对于 AX,BX,CX,DX 这四个寄存器来讲,可以单独存取它们的高 8 位和低 8 位 (AH,AL,BH,BL,CH,CL,DH,DL)。它们的含义如下：

```
EAX：累加器
EBX：基址寄存器
ECX：计数器
EDX：数据寄存器
ESI：源地址指针寄存器
EDI：目的地址指针寄存器
EBP：基址指针寄存器
ESP：堆栈指针寄存器
```

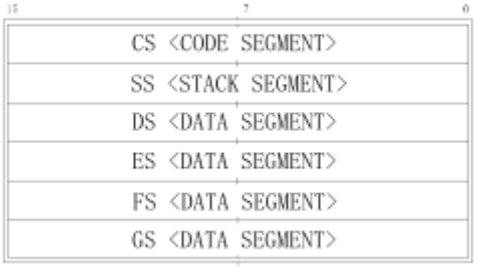
通用寄存器



Segment Register(段寄存器，也称 Segment Selector，段选择符，段选择子)：除了 8086 的 4 个段外(CS,DS,ES,SS)，80386 还增加了两个段 FS，GS,这些段寄存器都是 16 位的，用于不同属性内存段的寻址，它们的含义如下：

```
CS：代码段(Code Segment)
DS：数据段(Data Segment)
ES：附加数据段(Extra Segment)
SS：堆栈段(Stack Segment)
FS：附加段
GS 附加段
```

段寄存器

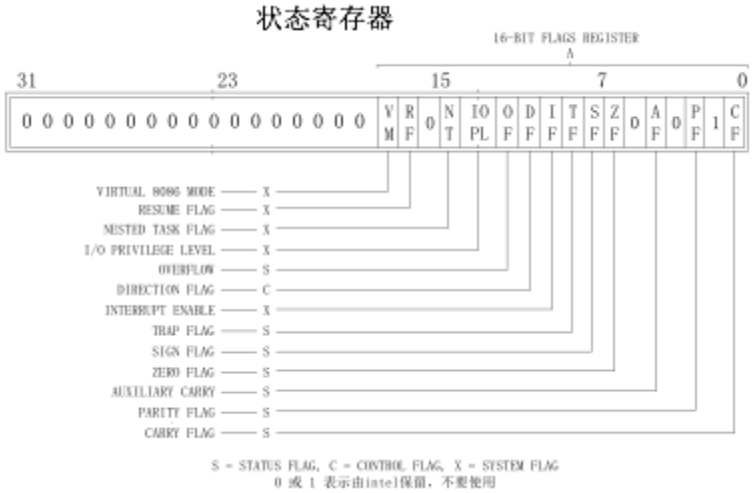


Instruction Pointer(指令指针寄存器)：EIP 的低 16 位就是 8086 的 IP，它存储的是下一条要执行指令的内存地址，在分段地址转换中，表示指令的段内偏移地址。

状态寄存器和指令寄存器

Flag Register(标志寄存器)：EFLAGS,和 8086 的 16 位标志寄存器相比，增加了 4 个控制位，这 20 位控制/标志位的位置如下图所示：

状态寄存器



相关的控制/标志位含义是:

CF(Carry Flag)：进位标志位；

PF(Parity Flag)：奇偶标志位；

AF(Assistant Flag)：辅助进位标志位；

ZF(Zero Flag)：零标志位；

SF(Singal Flag)：符号标志位；

IF(Interrupt Flag)：中断允许标志位,由 CLI，STI 两条指令来控制；设置 IF 位使 CPU 可识别外部（可屏蔽）中断请求，复位 IF 位则禁止中断，IF 位对不可屏蔽外部中断和故障中断的识别没有任何作用；

DF(Direction Flag)：向量标志位，由 CLD，STD 两条指令来控制；

OF(Overflow Flag)：溢出标志位；

IOPL(I/O Privilege Level)：I/O 特权级字段，它的宽度为 2 位,它指定了 I/O 指令的特权级。如果当前的特权级别在数值上小于或等于 IOPL，那么 I/O 指令可执行。否则，将发生一个保护性故障中断；

NT(Nested Task)：控制中断返回指令 IRET，它宽度为 1 位。若 NT=0，则用堆栈中保存的值恢复 EFLAGS，CS 和 EIP 从而实现中断返回；若 NT=1，则通过任务切换实现中断返回。在 ucore 中，设置 NT 为 0。

还有一些应用程序无法访问的控制寄存器，如 CR0,CR2，CR3...，将在后续章节逐一讲解。

BootLoader

Boot

Loader

内核雏形

上一章我说到，为了加载 ELF 格式的内核进内存，我们必须研究一下这种格式。

GCC 内联汇编

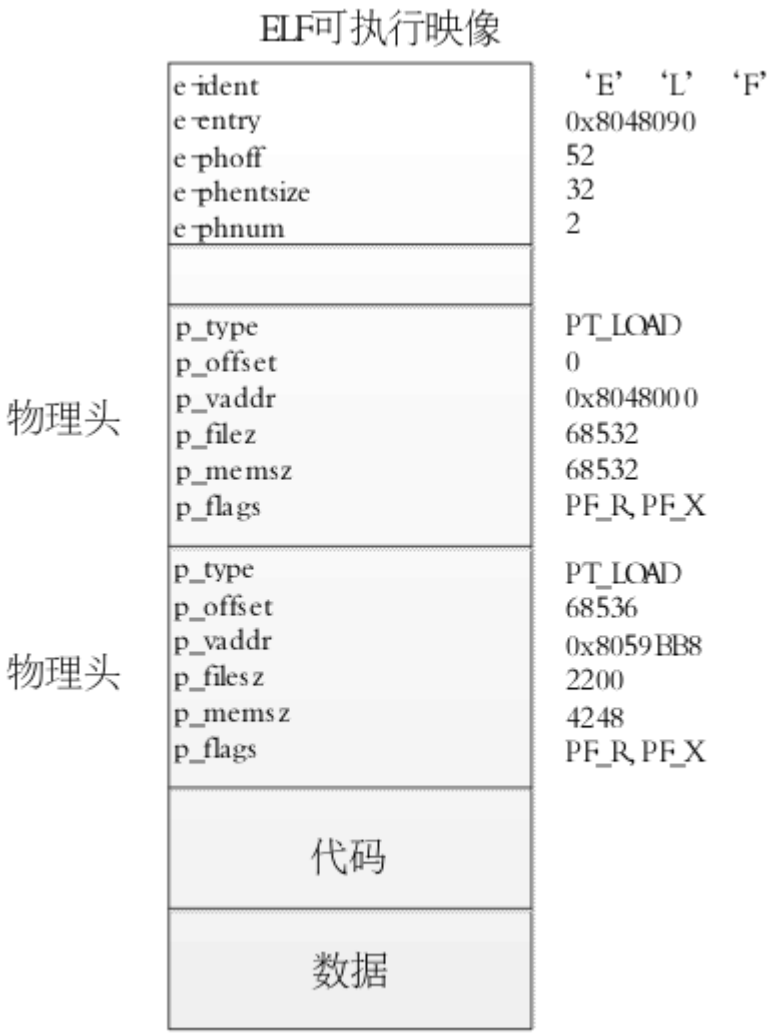
ELF

由于 bootloader 会访问 ELF(Executable and linking format)格式的 MOS，并把 MOS 加载到内存中。所以，在这里我们需要简单介绍一下 ELF 文件格式，以帮助我们理解 MOS 的整个编译、链接和加载的过程，特别是对 ld 链接器用到的链接地址（Link address）和操作系统相关的加载地址（Load address）要有清楚的了解。

ELF 文件格式是 Linux 系统下的一种常用目标文件(object file)格式，有三种主要类型。可重定位文件(relocatable file)类型和共享目标文件(shared object file)类型在本项目中没有涉及。本项目的 OS 文件类型是可执行文件(executable file)类型，这种 ELF 文件格式类型提供程序的进程映像，加载程序的内存地址描述等。

简单地说，bootloader 通过解析 ELF 格式的 MOS，可以了解到 MOS 的代码段（机器码）/数据段（初始化的变量）等在文件中的位置和大小，以及应该放到内存中的位置；可了解 MOS 的 BSS 段（未初始化的变量，具体内容没有保存在文件中）的内存位置和大小。这样 bootloader 就可以把 MOS 正确地放置到内存中，便于 MOS 的正确执行。

这里只分析与本项目相关的 ELF 可执行文件类型。ELF 的执行文件映像如下所示:



一个简单的ELF可执行文件的布局

ELF 的文件头包含整个执行文件的数据结构 elf header，描述了整个执行文件的组织结构。

```
struct elfhdr {
    uint32_t e_magic;     //ELF 文件格式魔数：必须等于 ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 标识文件类型：1=可重定位文件, 2=可执行文件, 3=共享目标文件, 4=内核镜像文件
    uint16_t e_machine;   // 标识程序运行的体系结构：3=x86, 4=68K, 等等.
    uint32_t e_version;   // 标识文件版本，总是为 1
    uint32_t e_entry;     // 可执行文件的入口地址
    uint32_t e_phoff;     // 程序头表在文件中的偏移量
    uint32_t e_shoff;     // 节头表在文件中的偏移量
    uint32_t e_flags;     // 特定体系结构标志,对 IA32 而言,此项为 0
    uint16_t e_ehsize;    // ELF 头大小
    uint16_t e_phentsize; // 程序头表中每一个条目的大小
    uint16_t e_phnum;     // 程序头表中条目个数
    uint16_t e_shentsize; // 节头表中每一个条目的大小
```

```
    uint16_t e_shnum;     // 节头表中条目个数
    uint16_t e_shstrndx;  // 包含节名称的字符串表是第几个节（从 0 开始数）
};
```

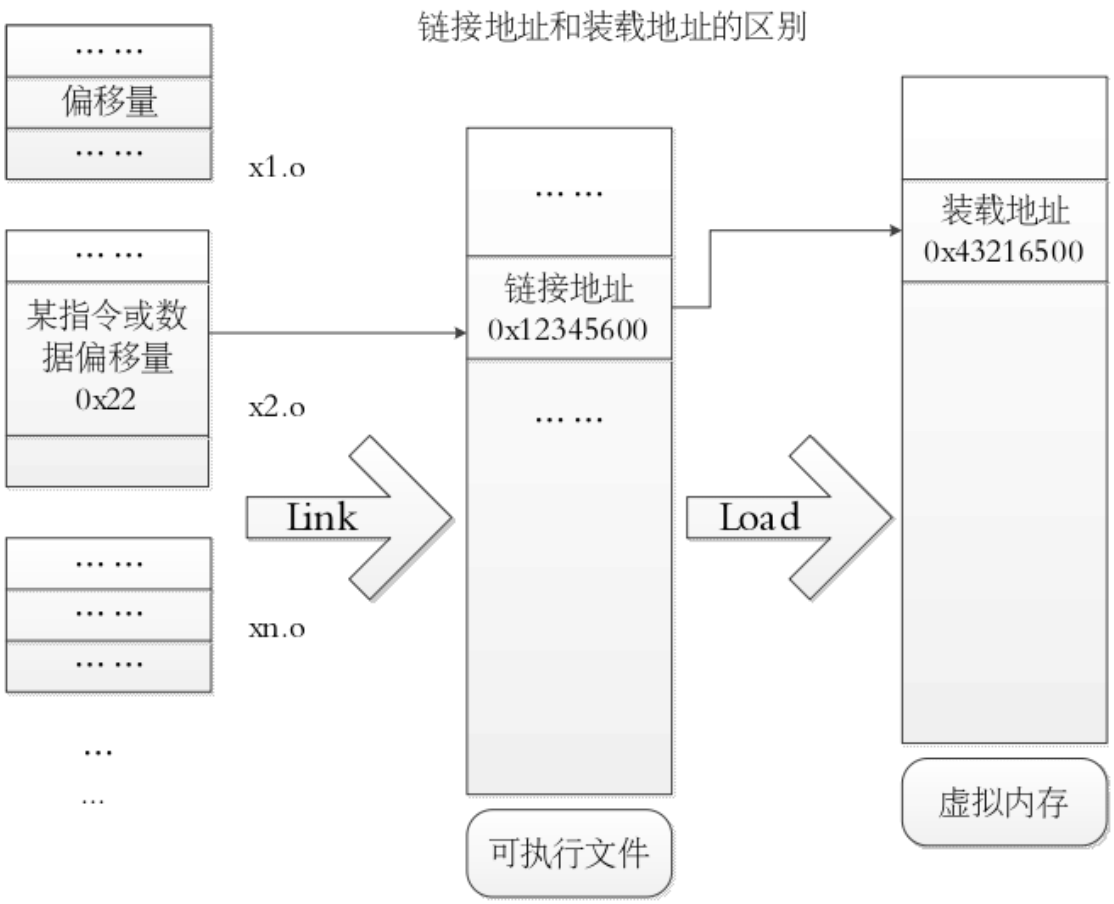
program header 描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序前面部分有一个 program header 结构的数组，每个结构描述了一个“段”(segment)或者准备程序执行所必需的其它信息。目标文件的“段”（segment） 包含一个或者多个“节区”（section），也就是“段内容（Segment Contents）”。program header 仅对于可执行文件和共享目标文件有意义。可执行目标文件在 elfhdr 的 e_phentsize 和 e_phnum 成员中给出其自身程序头部的大小。程序头部的数据结构如下表所示：

```
struct proghdr {
    uint32_t p_type;   // 当前程序头所描述的段的类型
    uint32_t p_offset; // 段的第一个字节在文件中的偏移
    uint32_t p_va;     // 段的第一个字节在内存中的虚拟地址
    uint32_t p_pa;     // 在物理地址定位相关的系统中，此项是为物理地址保留
    uint32_t p_filesz; // 段在文件中的长度
    uint32_t p_memsz;  // 段在内存中的长度
    uint32_t p_flags;  // 与段相关的标志
    uint32_t p_align;  // 根据此项值来确定段在文件以及内存中如何对齐
};
```

链接地址（Link address）和加载地址（Load address）：Link Address 是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address 是指程序被实际加载到内存的位置。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr 和 LoadAddr 不同会导致：

```
直接跳转位置错误
直接内存访问(只读数据区或 bss 等直接地址访问)错误
堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域
```

也存在 Link 地址和 Load 地址不一样的情况（如动态链接库）。在项目中，bootloader 和 MOS 的链接地址和加载地址是一致的。


链接地址和装载地址的区别
……
偏移量
……
x1.o
……
某指令或数据偏移量
0x22
x2.o
……
……
……
xn.o
…
…
Link
……
链接地址
0x12345600
……
可执行文件
Load
装载地址
0x43216500
虚拟内存

从 Loader 到内核

扩充内核

进程管理

输入/输出系统

键盘

键盘中断

AT、PS/2 键盘

键盘敲击过程分析

扫描码

键盘输入缓冲区

键盘操作处理任务

解析扫描码

显示器

TTY

基本概念

寄存器

TTY 任务

TTY 任务框架

多控制台

完善键盘处理

进程间通信

微内核

IPC

实现 IPC

文件系统

硬盘简介

硬盘驱动程序

硬盘分区表

遍历硬盘分区

FAT16 文件系统

FAT16 关键数据结构

文件系统接口

文件系统测试

将 TTY 纳入文件系统

内存管理

fork

exit 和 wait

exec

C 运行时库与简易 Shell

C 运行时库

简单用户程序

hello

pwd

echo

简易 Shell