## 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。

### 保护模式

## 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

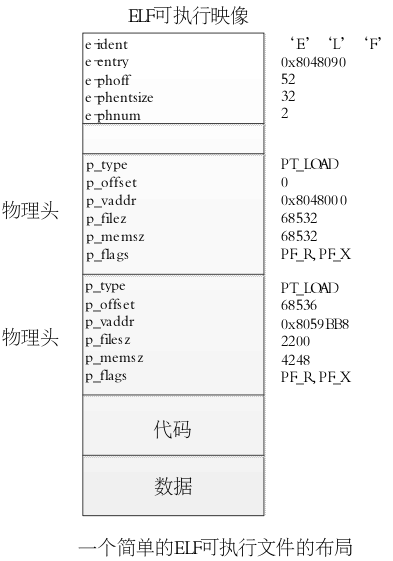
### ELF

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：



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

};

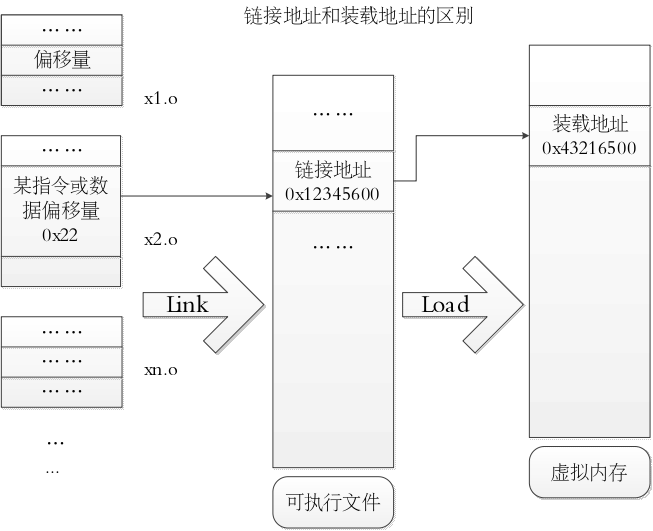
**链接地址（Link address）和加载地址（Load address）：**Link Address是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address是指程序被实际加载到内存的位置。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr和LoadAddr不同会导致：

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。



### 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