目录

Linux 考古笔记

2017 年 10 月，泰晓科技 发起并成立了“Linux 考古队”，志在共同学习和研究历史版本的 Linux 内核，输出学习笔记并汇总成册。

这个小册子就是用来记录“考古队”的各项考古发现并及时分享给社区。

简介

- 代码仓库
- 在线阅读
- 项目首页
- 项目计划
- 提交规范

报名

欢迎各位致力于研究操作系统的同学们加入“考古队”，报名请通过扫描下方二维码联系这次考古行动的项目经理 Keven。

编译

要编译本书，请使用 Markdown Lab。

纠错

如有发现任何疑问，欢迎：

- 提交 Issues
- 提交 Pull Requests

版权

更多

- C 语言编程透视
- Shell 编程范例
- 嵌入式 Linux 知识库(eLinux.org 中文版)

Linux 考古项目计划书

2017 年 10 月，泰晓科技 发起并成立了“Linux 考古队”，志在共同学习和研究历史版本的 Linux 内核，输出学习笔记并汇总成册。

项目概述

项目目标

- 完成对早期 Linux kernel 的探索和学习，掌握各个核心模块
- 输出研究学习文档，并汇总成册，PR 到 lad-book 并编译成书
- 输出研究学习成果，录制视频分享知识

项目开发环境

- Cloud Lab
- Linux 0.11
- Markdown Lab

应交付的成果

- 完整的内容梳理流程图
- 加过 debug 信息的代码
- 详细的结题报告（梳理文档）
- 对应的直播视频分享

项目验收机制

实行队长验收制，由各个模块的负责人进行 Check，然后提交 PR 到 PM，PM 核验完毕后合入 lad-book。

项目愿景

- 通过此次考古，各个团队争当 Linux Kernel 领域的专家团队
- 做中国开源在线协作领域的先行者，推动者。

团队组织

组织结构

角色	职责	备注
Manager	提供项目所需要的核心部件支持，包括核心实验环境。定期审阅项目经理提交的相关报告	
PM	制定项目计划并依据计划对项目进行监督和跟踪控制。定期发布 Release 并向经理汇报项目进展，对项目中出现的问题及时采取措施	
组长	研究，组织，协调，汇总学习成果。并定期组织内部电话会议，维护和校订组内输出成果并 PR，保持进度同步	
组员	负责各个模块的研究，学习，输出	.

人员分工

姓名	角色	职责	备注
吴章金	发起人	核心实验环境的提供者，项目发起人，开源领域专家，先行者	泰晓科技创始人，项目发起者
Keven	项目经理	帮助大家制定计划，辅助各模块负责人把控考古质量和成果	多年内核驱动开发经验，目前在知名外企担任 Kernel 开发工程师，项目推动者
史璞金	A 组开发组长	制定组内考古计划，输出文档，定期组织会议	211 高校大四学生，有丰富的社区经验以及强烈的责任感，在校参与多个项目开发
张灏	B 组开发组长	同上	多年嵌入式开发经验，熟悉各种芯片，目前担任驱动开发工程师
Keven	C 组开发组长	同上	同上上上
李松泽	D 组开发组长	同上	多年工作经验，目前在知名国内集成电路公司担任驱动开发工程师
王举利	E 组开发组长	同上	多年驱动开发工作经验，目前在知名外企担任驱动开发工程师
燕涛	F 组开发组长	同上	多年设备驱动开发经验，目前在互联网巨头（BAT 中最厉害的那个）从事虚拟化测试工程师
方英宁	G 组开发组长	同上	多年设备驱动开发经验，目前在国内知名半导体公司从事驱动开发经验
陈恩召	H 组开发组长	同上	近 10 年驱动开发经验,目前从事网路相关开发工作

小组成员

组长	小组编号	研究模块	内容描述	预设人数
史璞金	A	背景知识	C语言、汇编、数据结构	5
张灏	B	基础部分	内核体系结构(进程相关、堆栈相关、核心调度)	5
Keven	C	基础部分	内核体系结构(中断子系统、定时器、时钟、异常)	5
李松泽	D	引导和启动	引导、初始化过程	5
王举利	E	设备驱动	块设备和字符设备驱动	5
燕涛	F	文件系统	文件系统	5
方英宁	G	内存管理	内存管理	5
陈恩召	H	库文件，头文件，以及编译规则	头文件、库文件、编译规则	5

协作和沟通

线上内部协作

由于考古队成员来自全国各地，大家都是通过网络来进行沟通和学习。为了更好的进行协作，我们项目采用了 github + gitbook 的在线协作模型。 并采用微信群，邮件群等多样式的沟通工具。

线下沟通

考古队将不定期举办线下沙龙活动。

实施计划

工作流程

详情见上述实验环境

项目时间计划

任务	人员	开始时间	截止时间	是否结束
成立考古队	泰晓科技	2017.10.1	2017.10.9	是
创建 lad-book 及项目首页	吴章金	2017.10.10	2017.10.15	是
筛选组长	Keven	2017.10.20	2017.11.5	是
重新报名并建立小组群	各组组长	2017.11.4	2017.11.5	是
熟悉在线协作流程	所有人	2017.11.6	2017.11.13	是
确定规范	Keven	2017.11.9	2017.11.16	是
各组长组织各组阶段性输出输出	所有人	2017.11.16	2017.12.16	否

质量保证计划

实行互相 Review 的规则，组长将组内输出内容PR后，由 PM 将 PR 的内容简单审核，然后下发其他组组长进行二次 Review。

进度控制计划

将定期同步考古进度，把控每个组的考古计划。

清理计划

对于不参与的成员将进行清理，清理规则如下:

- 报名后，长时间没有输出
- 报名后，长时间没有分享

赞助

为了更好的推进这次考古活动，期待不能亲自参与的同学能够赞助我们。相关费用将用于设立项目奖，用来激励更多同学参与Linux 0.11 的学习和研究并撰写考古笔记。

赞助方式请通过泰晓服务中心 进行。

更多高质量的考古成果需要您的支持！

> lad Team

Lad Book 提交规范

章节目录介绍

- README.md - 本书简介
  - SUMMARY.md - 书目录
  - MAINTAINERS.md - 贡献者名单
  - plan.md - 项目计划
  - sumbit.md - 提交规范
- background - Linux 背景详解
  - history.md - Linux 历史
  - principle.md - 微机原理
  - programming - 内核编程
    - assembly.md - 汇编语言
    - c.md - C 语言
    - exec.md - 目标文件格式
    - makefile.md - Makefile
  - isa - ISA
    - x86.md - X86
    - arm.md - ARM
    - mips.md - MIPS
    - powerpc.md - PowerPC
  - emulator - 模拟器
    - qemu.md - Qemu
    - bochs.md - Bochs
- lab - 本书实验环境详解
  - env - 搭建环境
    - linux.md - Linux
    - windows.md - Windows
    - macosx.md - Mac OSX
  - experiment - 实验过程
    - edit.md - 编辑
    - compile.md - 编译
    - run.md - 运行
    - debug.md - 调试
  - filesystem - 文件系统
    - image.md - 镜像制作
    - share.md - 文件传输
  - source - 源码分析
    - calltree.md - 调用关系
    - index.md - 代码检索
- kernel - 内核体系结构详解
  - model.md - 内核模式
  - arch.md - 内核架构
  - core - 核心功能
    - interrupt.md - 中断机制
    - exception.md - 异常处理
    - timer.md - 时钟管理

        - process.md - 进程管理
        - mm.md - 内存管理
        - filesystem.md - 文件系统
        - stack.md - 堆栈用法
        - syscall.md - 系统调用
    - app.md - 应用程序
    - source.md - 源码结构
    - config.md- 配置系统
    - compile.md - 编译系统
- 0.11 - 0.11 内核详解
    - build.md - 建构工具
    - boot.md - 引导启动程序
    - init.md - 初始化程序
    - kernel.md - 内核代码
    - math.md - 数学协处理器
    - mm.md - 内存管理
    - filesystem.md - 文件系统
    - drivers - 驱动程序
        - block.md - 块设备
        - char.md - 字符设备
    - header.md - 头文件
    - lib.md - 库文件
- refs - 参考资料
    - standards - 标准文件
        - a.out.md - a.out
        - elf.md - ELF
        - ascii.md - ASCII 码表
        - posix.md - POSIX
    - books.md - 经典书籍
    - links.md - 重要网址
- images - 图片目录
    - CHAPTER - 本书章节图片
    - wechat - 本书贡献者微信

图片

上传图片

图片请根据上述目录结构统一存放在 `images/` 目录下，图片命名请以章节名为前缀，图片内容梗概为后缀，尽量言简意赅，不要过于冗长。

例如：`images/0.11/mm-arch.jpg` 。

制图工具

本地工具

- dia

在线工具

- Processon

提交指南

TODO: 增加演示链接

> Lad-book Team

考古人员名单

本章介绍相关背景知识，内容涵盖 Linux 发展的历史脉络、微机结构、内核编程、指令集架构以及处理器和板级模拟。

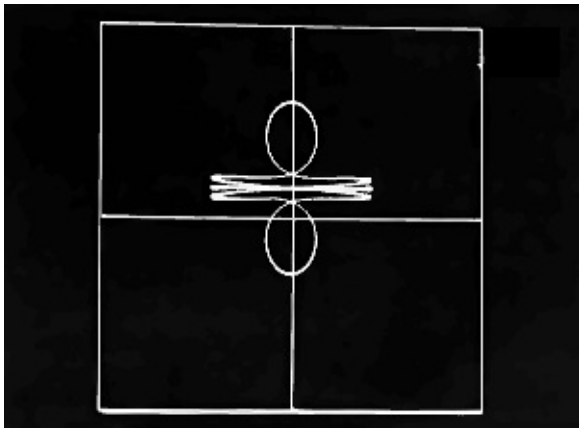
Linux 历史概述

Linux 是一种自由和开放源代码的类 UNIX 操作系统。也是目前运用领域最广泛、使用人数最多的操作系统。该操作系统的内核由 Linus Torvalds（林纳斯·托瓦兹）在 1991 年 10 月 5 日首次发布。

Linux 最初是作为支持英特尔 X86 架构的个人电脑的一个自由操作系统。目前 Linux 已经被移植到更多的计算机硬件平台，远远超出其他任何操作系统。Linux 可以运行在服务器和其他大型平台之上，如大型主机和超级计算机。Linux 也广泛应用在嵌入式系统上，如手机、平板电脑、路由器、电视和电子游戏机等。

Unix 的发展历史

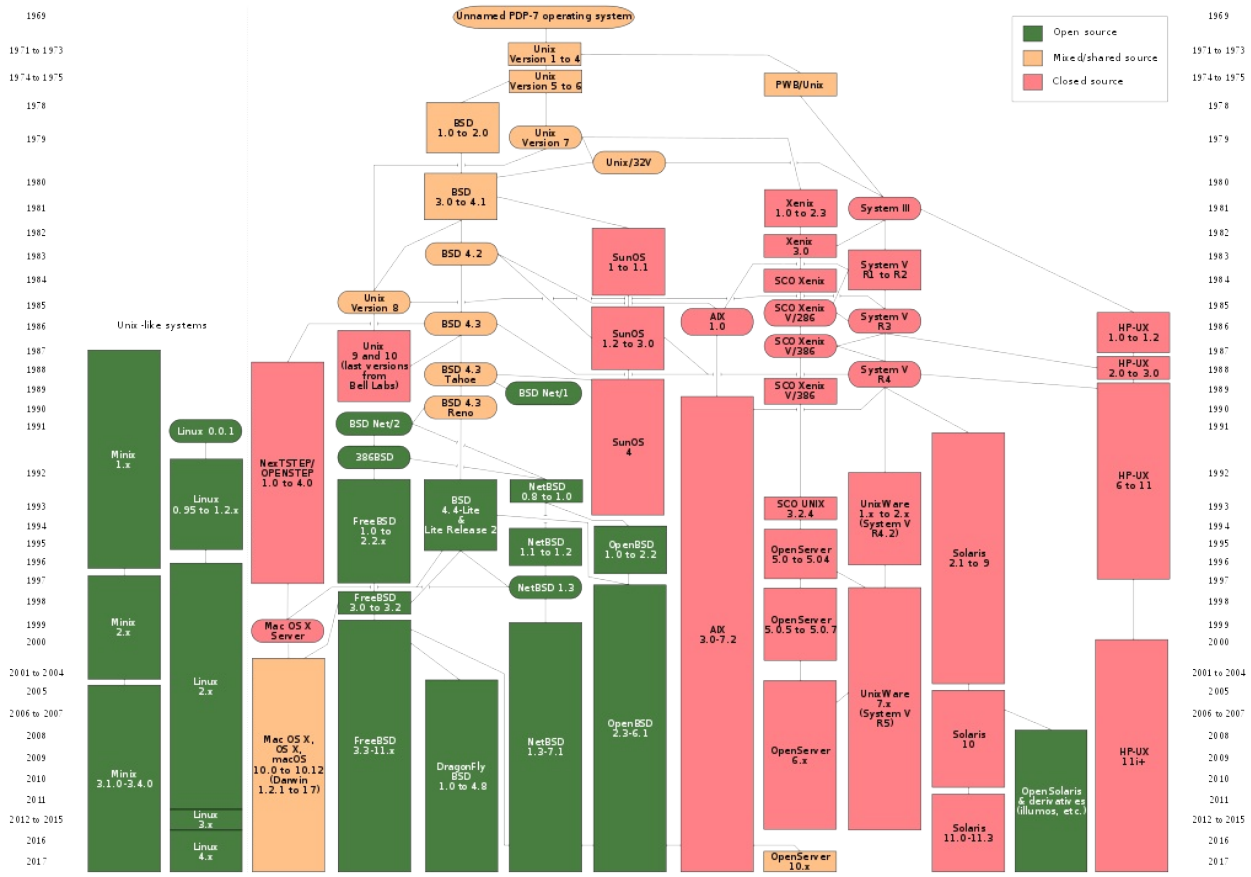
1969 年肯·汤普逊(Ken Thompson)在参与美国 AT&T 公司贝尔实验室的 Multics 操作系统项目的过程中开发了一款游戏——《星际旅行》。这是一款飞行模拟游戏。玩家需要控制太空飞船在黑色背景和白色线条组成的太阳系中飞行，并在不同行星和卫星之间着陆，没有特定的目标。



Multics 项目失败之后，为了能够在 GECOS 系统上继续玩这款游戏，他用 Fortran 语言重写了游戏代码。这款包含许多物理学知识的游戏成功的吸引了包括 Ravi Sethi 和 丹尼斯·里奇(Dennis Ritchie)等人的兴趣。在那个年代，玩游戏可谓代价高昂。因为贝尔实验室内多个终端连接一台中央电脑，终端只有输入/输出的功能，没有运算能力，每个终端要将任务提交到任务队列中，依次排队运行。一旦运行游戏，将会占用很长时间，其他任务只能暂停。所以，玩一次游戏的成本大约是 50 至 75 美元。后来汤普逊发现周围的部门有一台老旧且很少被使用的 PDP-7 小型机，于是又将游戏移植到新系统，但他发现游戏在新机器下运行很慢，于是又吸取了丹尼斯·里奇(Dennis Ritchie)和 Rudd Cassaway 在开发 Multics 文件系统时的经验，在他们工作的基础上设计了自己的文件系统。后经扩展，形成了一个完备的操作系统，在公司内部广泛传播，并于 1970 年被命名为 Unix。

1973 年，Unix 被丹尼斯·里奇用 C 语言（内核和 I/O 除外）重新编写。高级语言编写的操作系统具有更佳的兼容性，也能更容易地移植到不同的计算机平台。

由于 Unix 的高度可移植性、强大的效能、简洁的系统设计、开放的源代码等特点，吸引了许多其他组织和团体对其进行了扩展，最著名当属加州伯克利分校。他们推出了 Berkeley Software Distributions（BSD）。BSD 系统的主要特性包括： vi 文本编辑器 (ex 可视模式) 、C shell 和 TCP/IP 的早期实现版等，这些特性至今仍被沿用。此外，很多商业公司开始了 Unix 操作系统的衍生开发，例如 AT&T 自家的 System V、IBM 的 AIX 以及 HP 与 DEC 等公司，都有推出自家的主机搭配自己的类 Unix 操作系统。但是，每家公司推出的硬件架构不同，加上当时没有所谓的协议的概念，导致开发出来的 Unix 操作系统以及内含的相关软件并没有办法在其他的硬件架构下工作！1979 年，AT&T 推出 System V 第七版 Unix 后，这个情况就有点改善了。 这一版最重要的特色是可以支持 X86 架构的个人计算机系统，也就是说 System V 可以在个人计算机上面安装与运作了。



由于商业的考虑，以及在当时现实环境下的思考， AT&T 在 1979 年发行的 System V 第七版 Unix 中，特别提到了『不可对学生提供源代码』的严格限制！同时，也造成 Unix 业界之间的紧张气氛，并且也引爆了很多的商业纠纷。

MINIX 操作系统

由于 1979 年的版权声明的影响，在大学中不能使用Unix源代码用于教学，促使 Andrew Tanenbaum （谭宁邦）教授自己动手开始编写用于教育用途的 Minix 这个类 Unix 的核心程序。在撰写的过程中，为了避免版权纠纷，谭宁邦完全不看 Unix 核心原代码！并且强调他编写的 Minix 系统与 Unix 系统兼容！到 1986 年 Minix 开发工作终于完成。Minix 并不是完全免费的，无法在网络上提供下载，必须要通过磁盘购买才行，所以 Minix 的传递速度并没有很快速！此外，购买时，随磁盘还会附上 Minix 的源代码！

Minix 除启动部分使用汇编语言编写外，其余部分都采用 C 语言编写。共约 12000 行代码，并作为示例放置在他的著作《操作系统：设计与实现》一书的附录当中。值得一提的是，2015 年之后发布的英特尔芯片都在内部都运行着 MINIX 3，作为 Intel 管理引擎(Intel Management Engine)的组件。

GNU 计划

GNU 计划，是一个自由软件集体协作项目。1983 年 9 月 27 日由 Richard Mathew Stallman（理查德·斯托曼）在麻省理工学院公开发起。它的目标是创建一套完全自由的操作系统。GNU 是 “GNU is Not Unix” 的递归缩写。

到了1985年，为了避免GNU所开发的自由软件被其他人所利用而成为专利软件， 所以他与律师草拟了有名的通用公共许可证(General Public License, GPL)， 并且称呼为 copyleft (相对于专利软件的 copyright！)。

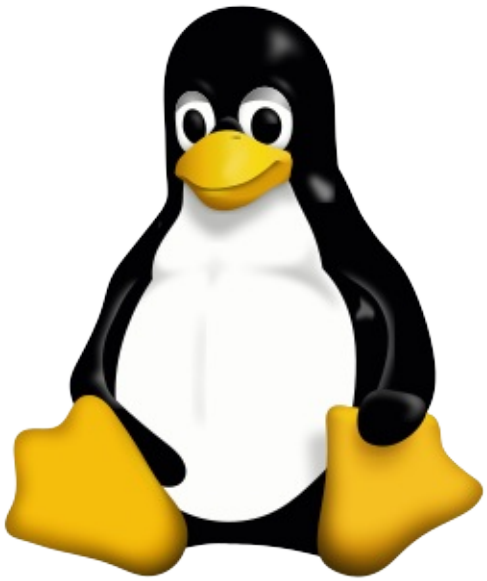
GNU 项目开发出许多高质量的免费软件，其中包括有名的 Emacs 编辑系统、Bash Shell 程序、Gcc 系列编译程序、Gdb 调试程序等等。不过，对于 GNU 的最初构想『建立一个自由的 Unix 操作系统』来说，有这些优秀的程序是仍无法满足的，因为，当时并没有『自由的 Unix 核心』存在。所以这些软件仍只能在那些有专利的 Unix 平台上工作，一直到 Linux 的出现。

Linux 的发展史

1988 年，Linus Torvalds（林纳斯·托瓦兹）进入了赫尔辛基大学，并选读了计算机科学系。在读期间，接触到了 Unix 系统。当时整个赫尔辛基只有一部最新的 Unix 系统，同时仅提供16个终端（terminal）。早期的计算机仅有主机具有运算功能，terminal 仅负责提供 Input/Output 而已。在这种情况下，实在很难满足托瓦兹的需求，不久之后，他就知道有一个类似 Unix 的系统，并且与 Unix 完全兼容，还可以在 Intel 386 机器上面运行的操作系统——Minix。托瓦兹贷款购买了一台 Intel 386 个人计算机后，便安装了 Minix 操作系统，并通过 Minix 操作系统附带的源代码，学习到了很多内核程序设计的理念。

由 Andrew Tanenbaum（谭宁邦）教授只愿意将 Minix 系统用于教学，不愿意强化系统功能，导致大部分工程师不满足于已有的功能，也使托瓦兹萌生自己编写操作系统的想法。而 GNU 计划产出的 Bash 工作环境和 Gcc 编译程序等自由软件，让托瓦兹能够顺利的编写内核程序。最终，在 1991 年，他写出了能够在 386 计算机上运行的内核，并将其上传到网络上，提供下载。为了让更多 Unix 系统上的软件运行在 Linux 上，托瓦兹参考 POSIX 标准规范修改 Linux，提高了与 Unix 系统的兼容性。

1996年，Torvalds 为 Linux 选定了企鹅作为它的吉祥物。Larry Ewing 提供了吉祥物的初稿。现在正在使用的著名的吉祥物就是基于这份初稿的。James Hughes 根据“Torvalds's Unix”的谐音，取名为 Tux。



主要版本

Linux 操作系统从诞生到 1.0 版正式出现，共发布了下表中所示的一些主要版本（引用自《Linux内核完全剖析》赵炯编著的第五页）。

版本号	发布/编制日期	说明
0.00	1991.2～4	两个进程，分别在屏幕上显示“AAA...”和“BBB...”（注:没有发布）
0.01	1991.9.17	第一个正式向外公布的 Linux 内核版本。多线程文件系统、分段和分页内存管理。还不包含软盘驱动程序
0.02	1991.10.5	该版本以及 0.03 版是内部版本，目前已经无法找到。特点同上
0.10	1991.10	由 Ted Ts'o 发布的 Linux 内核版本。增加了内存分配库函数。在 boot 目录中含有一个把 as86 汇编语法转换成 Gas 汇编语法的脚本程序
0.12	1992.1.15	主要增加了数学协处理器的软件模拟程序。增加了作业控制、虚拟控制台、文件符号链接和虚拟内存对换（swapping）功能
0.95.x（即0.13）	1992.3.8	加人虚拟文件系统支持，但还是只包含一个 MINIX 文件系统。增加了登录功能。改善了软盘驱动程序和文件系统的性能。改变了硬盘命名和编号方式。原命名方式与 MINIX 系统的相同，此时改成与现在 Linux 系统的相同。支持 CDROM
0.96.x	1992.5.12	开始加人UNIX socket支持。增加了 ext文件系统测试程序。 I驱动程序被正式加人内核。软盘类型自动识别。改善了串行驱动、高速缓冲、内存管理的性能，支持动态链接库，并开始能运行 x· Wind“程序。原汇编语言编制的键盘驱动程序已用 c重写。与0，95内核代码比较有很大的修改
0.97.x	1992.8.1	增加了对新的 SCSI 驱动程序的支持；动态高速缓冲功能；msdos 和 ext 文件系统支持；总线鼠标驱动程序。内核被映射到线性地址 3GB 开始处
0.98.x	1992.9.28	改善对 TCP/IP（0.8.1）网络的支持，纠正了 extfs 的错误。重写了内存管理部分（mm），每个进程有 4GB 逻辑地址空间（内核占用 1GB）。从 0.98.4 开始每个进程可同时打开 256 个文件（原来是32个），并且进程的内核堆栈独立使用一个内存页面
0.99.x	1992.12.13	重新设计进程对内存的使用分配，每个进程有 4GB 线性空间
1.0	1994.3.14	第一个正式版本

Linux的核心版本命名

目前最新的内核版本是 2017 年 11 月 24 日公布的 4.14.2 版，Linux 内核版本号命名的规则：r.x.y

- r：表示目前发布的内核主版本。
- x：x是偶数表示稳定版本，主要用于企业等生产环境; 若是奇数表示开发中版本，主要用于开发和测试新功能。
- y: 修订版本号，表示修改的次数。

名称争议

GNU 计划的支持者与开发者，特别是其创立者 Richard Matthew Stallman（理查德·斯托曼）主张 Linux 应称为“GNU/Linux”较为恰当，因为此类操作系统使用了众多 GNU 程序，包含Bash（Shell 程序）、库、编译器等等作为 Linux 内核上的系统包。Linux 社区中的一些成员，如Eric Steven Raymond（埃里克·斯蒂芬·雷蒙）、Linus等人，偏好 Linux 的名称，认为 Linux 朗朗上口，短而好记，拒绝使用“GNU/Linux”作为操作系统名称。并且认为 Linux 并不属于 GNU 计划的一部分。现在，有部分 Linux 发行版，如 Debian ，采用了“GNU/Linux”的称呼。

参考资料

- Linux 内核完全剖析[M],赵炯,机械工业出版社,2016
- Linux 内核设计的艺术:图解 Linux 操作系统架构设计与实现原理[M],新设计团队,机械工业出版社,2013
- 鸟哥的 Linux 私房菜 基础学习篇[M],鸟哥,人民邮电出版社,2010
- Linux EB/OL, https://zh.wikipedia.org/zh-cn/Linux
- Space Travel EB/OL, https://zh.wikipedia.org/wiki/%E6%98%9F%E9%99%85%E6%97%85%E8%A1%8C_(1969%E5%B9%B4%E6%B8%B8%E6%88%8F)
- BSD EB/OL, https://zh.wikipedia.org/zh-cn/BSD
- Minix EB/OL, https://zh.wikipedia.org/zh-cn/Minix
- History_of_Linux EB/OLx, https://en.wikipedia.org/wiki/History_of_Linux

本章介绍 Linux 考古所需的实验环境。

Linux 0.11 Lab 是一个基于 Docker/Qemu/Bochs 的操作系统实验环境，早期为 Linux 0.11 定制，未来会不断加入更多老版本的 Linux 内核，方便大家通过回顾 Linux 内核的历史演变，结合动手实践，进而更深入地理解操作系统原理。

本章介绍 Linux 内核的体系架构。

本章介绍 Linux 内核的核心功能。

异常处理

在 Linux 0.11 的代码树中，内核异常处理的服务程序分别是 trap.c 和 panic.c，trap.c 负责处理硬件异常，panic.c 则实现了内核的异常处理接口。

上面提到了中断和异常，那么两者有什么异同呢？

从理论的角度我们可以笼统的说中断是指中央处理器（CPU）对系统发生某个事情后作出的一种反应，异常的定义没有明确的规定，不同的体系架构的定义有一些差异，不过大体上可以认为异常是由于软件造成的。

一般情况下，一个完整的可用操作系统由 4 部分组成。分别是硬件、操作系统内核、操作系统服务以及用户应用程序。

当前 Linux 0.11 代码树是 Linus 基于 Intel 的 386 兼容机编写的。其 CPU 为80386，想一下，假如你是 Linus，你要为你的操作系统来适配 386 兼容机，现在你要完成异常处理部分，你应该做什么？

答案毋庸置疑吧，所以具体关于中断和异常的处理我们来看看 CPU 手册中是如何描述的。那么，我们先按照一个正常开发流程去模拟当前 Linus 是如何开发异常处理模块的。

80386

在 i386 的数据手册中，有一章节描述了中断和异常。

原文是这么描述的:

> The 80386 has two mechanisms for interrupting program execution:
>
> ```
> 1. Exceptions are synchronous events that are the responses of the CPU to
> certain conditions detected during the execution of an instruction.
>
> 2. Interrupts are asynchronous events.
> ```
>
> Interrupts and exceptions are alike in that both cause the processor to temporarily suspend its present program execution in order to execute a program of higher priority.
>
> The major distinction between these two kinds of interrupts is their origin. An exception is always reproducible by re-executing with the program and data that caused the exception, whereas an interrupt is generally independent of the currently executing program.
>
> Application programmers are not normally concerned with servicing interrupts. More information on interrupts for systems programmers may be found in Chapter 9. Certain exceptions, however,are of interest to applications programmers,and many operating systems give applications programs the opportunity to service these exceptions. However,the operating system itself defines the interface between the applications programs and the exception mechanism of the 80386.

看手册找重点，我们来总结一下上述描述的一些关键点。

- 80386 为中断程序执行提供了两种机制
  - 异常是同步事件，用于响应指令执行过程检测到的特定条件。
  - 中断是异常事件，是由外部设备触发。
- 中断和异常异同点
  - 两者都会导致的是：CPU 暂停当前程序的执行，去处理优先级更高的程序。
  - 异常可以重复触发，只要用同样的程序和数据反复执行。
  - 中断则不然，它通常是独立于当前执行的程序的。
- 应用开发人员通常不关心服务的中断。
- 应用开发人员只需使用操作系统提供的应用接口，这些接口的使用不当可能也会产生异常。

关于中断机制的详细内容，本章节不再赘述。请阅读本书中断处理机制章节内容。

上面大抵描述了 80386 的异常和中断概念。接下来继续阅读手册，看看 80386 到底提供了哪些详细的异常事件?

80386 异常向量表

向量偏移值	描 述	说 明
0	Devide Error	当进行除以零的操作时产生
1	Debug Exceptions	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的T标志时产生这个中断
2	NMI Interrupt	由不可屏蔽产生
3	Breakpoint	由断点指令INT3产生，与 Debug 处理相同
4	INTO Detected Overflow	eflags 的溢出标志 0F 引起
5	BOUND Range Exceeded	寻址到有效地址以外引起
6	Invalid Opcode	CPU执行发现一个无效的指令操作码
7	Coprocessor Not Available	设备不存在，指协处理器。在两种情况下会产生该中断：a：CPU 遇到一个转意指令并且EM置位。b：MP 和 TS 都在置位状态， CPU 遇到wait或一个转意指令。在这种情况下，处理程序在必要应该更新协处理器的状态
8	Double Exception	双故障出错
9	Coprocessor Segment Overrun	协处理器段超出
10	Invalid Task State Segment	CPU 切换时发现 TSS 无效
11	Segment Not Present	描述符所指的段不存在
12	Stack Fault	堆栈溢出或者不存在
13	General Protection	没有符合 80386 保护机制的(特权机制)操作引起
14	Page Fault	页溢出或不存在
15	(reserved)	保留位
16	Coprocessor Error	协处理器检测到非法操作
17-32	(reserved)	保留位

上述表格便是 80386 手册中给出的所有异常向量，知道了每个异常的偏移地址，那么下面可以为内核写一个异常处理模块了，等等。好像还少一点什么？对，怎么访问向量表呢？

访问异常向量表

整个 CPU 域地址空间的划分请参考内存管理章节。

通过上述章节我们可以很清晰的获取 CPU 地址域的布局，并且知道了异常向量的 Base Addr。

```
val = ;
```

设计对应的内核数据结构

有了访问地址，异常向量偏移，那么我们就可以设计代码框架了。我想当时 Linus 应该是这么想的：我一直在研究 Unix 操作系统设计，所以我是否也可以将信号用到 Linux 操作系统中呢，可以把每个异常对应一个信号，用来做全局通知链，这样一些无需 CPU reset 解决的，可以做一下告警，告知开发应用程序或者驱动的程序员该如何规范使用当前芯片。好吧，我要开干了！！！

Linus 编写的异常代码

```
/*
 *  linux/kernel/traps.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */

/* 我们保存了一些硬件状态在'asm.s'中，然后我们使用'Traps.c'用来处理陷进和故障。当前主要用于调试，
 * 后续我们将扩展使其可以杀死一些令人厌恶的进程（或许可以给它一个信号，但是尽量还是将其杀死）。
 */

#include <string.h>

#include <linux/head.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/segment.h>
#include <asm/io.h>

/* 以下语句定义了三个嵌入式汇编宏语句函数，有关嵌入式汇编的基本语法见本程序列表后的说明。
 * 用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用,其中最后的__res是其输出值。
 * 第23行定义了一个寄存器变量__res，该变量将被保存在一个寄存器中，以便于快速访问和操作。
 * 如果想指定寄存器（例如eax），那么我们可以把该句写成register char __res asm("ax");
 * 取段seg中地址addr处的一个字节。
 * 参数：seg - 段选择符；addr - 段内指定地址。
 * 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))
 */

#define get_seg_byte(seg,addr) ({ \
register char __res; \
__asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
    :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

/* 取段seg中地址addr处的一个长字（4字节）。
 * 参数：seg - 段选择符；addr - 段内指定地址。
 * 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))
 */
```

```
#define get_seg_long(seg,addr) ({ \
register unsigned long __res; \
__asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
    :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

/* 取fs段寄存器的值(选择符)
 * 输出：%0 - eax (__res
 */

#define _fs() ({ \
register unsigned short __res; \
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

/* 下面是异常向量表对应的函数原型 */

int do_exit(long code);

void page_exception(void);

void divide_error(void);
void debug(void);
void nmi(void);
void int3(void);
void overflow(void);
void bounds(void);
void invalid_op(void);
void device_not_available(void);
void double_fault(void);
void coprocessor_segment_overrun(void);
void invalid_TSS(void);
void segment_not_present(void);
void stack_segment(void);
void general_protection(void);
void page_fault(void);
void coprocessor_error(void);
void reserved(void);
void parallel_interrupt(void);
void irq13(void);

/* die 函数主要用于打印出错的中断的名称、出错号、调用程序的EIP、EFLAGS、ESP、fs
 * 段寄存器值以及段的基址、段的长度、进程号pid、任务号、10字节指令码。
 * 如果堆栈在用户数据段，则还打印16字节的堆栈内容。
 */

static void die(char * str,long esp_ptr,long nr)
{
    long * esp = (long *) esp_ptr;
    int i;

    printk("%s: %04x\n\r",str,nr&0xffff);
    printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
        esp[1],esp[0],esp[2],esp[4],esp[3]);
    printk("fs: %04x\n",_fs());
    printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
    if (esp[4] == 0x17) {
        printk("Stack: ");
        for (i=0;i<4;i++)
            printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
        printk("\n");
    }
    str(i);
    printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
    for(i=0;i<10;i++)
        printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
    printk("\n\r");
    do_exit(11);        /* play segment exception */
}

/* 以下这些以 do_ 开头的函数是 asm.s 中对应中断处理程序调用的C函数。
```

```
void do_double_fault(long esp, long error_code)
{
    die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
    die("general protection",esp,error_code);
}

void do_divide_error(long esp, long error_code)
{
    die("divide error",esp,error_code);
}

void do_int3(long * esp, long error_code,
        long fs,long es,long ds,
        long ebp,long esi,long edi,
        long edx,long ecx,long ebx,long eax)
{
    int tr;

    __asm__("str %%ax":"=a" (tr):"0" (0));
    printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
        eax,ebx,ecx,edx);
    printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
        esi,edi,ebp,(long) esp);
    printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
        ds,es,fs,tr);
    printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

void do_nmi(long esp, long error_code)
{
    die("nmi",esp,error_code);
}

void do_debug(long esp, long error_code)
{
    die("debug",esp,error_code);
}

void do_overflow(long esp, long error_code)
{
    die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
    die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
    die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
    die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
    die("coprocessor segment overrun",esp,error_code);
}

void do_invalid_TSS(long esp,long error_code)
{
    die("invalid TSS",esp,error_code);
}
```

```
void do_segment_not_present(long esp,long error_code)
{
    die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
    die("stack segment",esp,error_code);
}

void do_coprocessor_error(long esp, long error_code)
{
    if (last_task_used_math != current)
        return;
    die("coprocessor error",esp,error_code);
}

void do_reserved(long esp, long error_code)
{
    die("reserved (15,17-47) error",esp,error_code);
}

/* trap_init 函数是异常（陷阱）中断程序初始化子程序，主要设置它们的中断向量表。
 * set_trap_gate() 与 set_system_gate() 都使用了中断描述符表IDT中的陷阱门（Trap Gate）
 * 它们之间的主要区别在于前者设置的特权级为0，后者是3。因此断点陷阱中断int3、溢出中断
 * overflow 和边界出错中断 bounds 可以由任何程序调用。这两个函数均是嵌入式汇编宏程序。
 * 更多可以查看 include/asm/system.h；第36行以及39行。
 */

void trap_init(void)
{
    int i;

    set_trap_gate(0,&divide_error);
    set_trap_gate(1,&debug);
    set_trap_gate(2,&nmi);
    set_system_gate(3,&int3);    /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
    set_trap_gate(8,&double_fault);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_trap_gate(14,&page_fault);
    set_trap_gate(15,&reserved);
    set_trap_gate(16,&coprocessor_error);
    for (i=17;i<48;i++)
        set_trap_gate(i,&reserved);
    set_trap_gate(45,&irq13);
    outb_p(inb_p(0x21)&0xfb,0x21);
    outb(inb_p(0xA1)&0xdf,0xA1);
    set_trap_gate(39,&parallel_interrupt);
}
```

panic

异常接口已经处理完毕，基本上触发上述事件，都有做处理。等等，假如出现很严重的事情怎么办呢？已经完全影响系统的关键部件的完整度了。Linus 当时可能是这么想的：我需要做一些事情，告知系统以及管理员，告知他们操作系统挂掉了，好吧，我先简单做一个接口吧，起个什么名字呢？panic，这个名字好像不错，就叫它吧。接口我先简单处理一下，假如我发现当前进程是 0 号进程，那么我就认为操作系统还没有挂载文件系统，是 swapper 进程出错了，反之不是的话我需要同步一下文件系统，然后制造死机吧（死循环）。

Linus 编写的panic 代码

```
/*
 *  linux/kernel/panic.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This function is used through-out the kernel (includeinh mm and fs)
 * to indicate a major problem.
 */
#define PANIC

#include <linux/kernel.h>
#include <linux/sched.h>

void sys_sync(void);    /* it's really int */

/* panic 函数用来打印内核中出现的重大错误信息，并运行文件系统同步函数，然后进入死循环。
 * 如果当前进程是任务0的话，还说明是交换任务出错，并且还没有运行文件系统同步函数。
 * 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一些的
 * 代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
 * 等同于现在gcc的函数属性说明：void panic(const char *s) __attribute__ ((noreturn));
 */

void panic(const char * s)
{
    printk("Kernel panic: %s\n\r",s);
    if (current == task[0])
        printk("In swapper task - not syncing\n\r");
    else
        sys_sync();
    for(;;);
}
```

测试

代码写完了，按照软件开发流程，下面就需要测试了，让我们编写测试用例触发上述异常（有的无法触发需要硬件配置），来验证效果吧。 由于篇幅较长，本书这里只简单演示一个测试用例，更多的测试用例，请在本书所带的资料中获取。

我们来验证堆栈溢出后内核发生什么？

测试代码

```

```

编译

运行

结果

在线协作

BitKeeper 是垃圾，先忍忍吧，现在贡献的人还不是很多，等我有空我会在开发一个协作工具。

参考资料

- INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986
- Linux 内核完全注释, 赵炯, 机械工业出版社, 2004

本章旨在分析 Linux 0.11 内核源码。

本章介绍本书需要用到的一些参考资料，包括相关标准、书籍和网址等。

- 《Linux 内核 0.11 完全注释》