NNOS 参考手册（第 0 版）

年年软件

二〇二〇年一月五日

扉页

[NNOS 初期根据川合秀实先生的《30 天自制操作系统》设计开发，谨谢。]

一开始面面俱到，往往寸步难行。

[GitHub 地址] https://github.com/nnrj/nnos
[NNOS 开发者交流群] 757581301
[开发团队]年年软件(BUG)、CPU。(时刻期待、欢迎新成员加入我们)。

本项目已通过 GPL 协议开源。

目录

〇 绪论 ……6

一 文档约定 ……7

1.1 约定 ……7
1.2 术语 ……7
1.3 快速运行指南 ……7

二 系统概述 ……8

2.1 CPU 架构 ……8
2.2 模块划分 ……8
2.3 功能框架 ……8
2.3.1 内存管理系统概述……9
2.3.2 I/O 管理系统概述……9
2.3.3 进程管理系统概述……9
2.3.4 文件管理系统概述……9
2.4 层次框架 …… 10
2.4.1 物理层……10
2.4.2 引导层……11
2.4.3 基础层……12
2.4.4 核心层……14
2.4.5 扩展层……16
2.4.6 应用层……17
2.5 整体流程 …… 17
2.5.1 IPL 运行流程……17
2.5.2 系统实模式运行流程……18
2.5.3 保护模式运行流程……19

三 详细参数 …… 19

3.1 项目源文件结构 …… 19
3.2 系统常量定义 …… 20
3.2.1 宏定义……20
3.2.2 静态数组……20
3.2.3 专用类型定义……21
3.3 系统结构体定义 …… 21
3.3.1 GDT 结构体和 IDT 结构体……21
3.3.2 FIFO 缓冲区结构体……22
3.3.3 BOOT 信息结构体……23
3.3.4 鼠标指针结构体……23
3.3.5 内存相关结构体……23
3.3.6 定时器相关结构体……24
3.3.7 图形相关结构体……24
3.3.8 图层相关结构体……25

3.3.9 任务相关结构体.....25
3.3.10 文件相关结构体.....26
3.4 引导层详细设计 ····· 26
3.4.1 IPL 详细设计.....26
3.4.2 SYSHEAD 详细设计.....28
3.4.3 OSFUN 详细设计.....29
3.5 基础层详细设计 ····· 29
3.5.1 GDT&IDT 详细设计.....29
3.5.2 INTERRUPT 详细设计.....30
3.5.3 TIMER 详细设计.....30
3.5.4 FIFO 详细设计.....30
3.5.5 Manager Interfac 详细设计.....30
3.6 核心层详细设计 ····· 31
3.6.1 Memery Manager 详细设计.....31
3.6.2 I/O Manager 详细设计.....32
3.6.3 Process Manager 详细设计.....33
3.6.4 FileManager 详细设计.....33
3.6.5 Graphics 详细设计.....34
3.6.6 BOOT 详细设计.....35
3.7 扩展层详细设计 ····· 36
3.8 应用层详细设计 ····· 36
3.8.1 Console 详细设计.....36
3.8.2 其他应用详细设计.....37
3.8.3 系统 U 盘启动详细设计.....37
四 测试分析 ····· 38
4.1 版本迭代测试 ····· 38
4.2 定时器性能测试 ····· 42
4.3 内核漏洞列表及详情 ····· 44
4.3.1 显存污染漏洞.....44
4.3.2 中断处理器无响应漏洞.....44
4.3.3 中断处理崩溃无限重启漏洞.....44
4.3.4 缓冲区溢出漏洞.....45
4.3.5 多任务死机漏洞.....45
4.3.6 文件信息区内存污染漏洞.....45
4.3.7 一般异常处理死循环.....46
五 开发环境搭建 ····· 46
5.1.1 开发平台及开发环境.....46
5.1.2 编程语言.....46
5.1.3 编译器.....47
5.1.4 虚拟机.....47
4.1.5 文件生成规则.....48
5.1 环境和工具列表 ····· 49
5.1.1 操作系统和硬件要求.....49

5.1.2 工具软件列表.....49
5.2 工具简介 49
5.2.1 开发环境搭建.....49
5.2.4 开发工具包详情.....51
5.2.5 编辑器和阅读器.....52
附录..... 55
附录I 硬件相关规定..... 55
附录II 项目源文件列表..... 57
附录III 系统常量表..... 59
附录IV 系统颜色信息注册表 68
附录V 键盘字符映射表..... 69
附录VI 系统函数简表..... 70
附录VII 系统 API 简表 77
附录VIII 系统升级日志..... 78
参考文献..... 82

○ 绪论

项目目标：自己动手实际地开发一个操作系统。
项目意义：兴趣、研究、学习。

1946 年首台通用电子计算机诞生后，软硬件迅速迭代。
二者相互促进。发展成今日精密稳定、复杂便捷的各类计算机系统。
在人类对系统资源利用率的极致追求下，操作系统应运而生。
期间，软硬件地位亦陡然变化。硬件日渐低廉，软件持续高昂
相对于硬件，操作系统已是计算机之魂。

诚然，Windows、GUN/Linux 等操作系统已极为成熟，一时难以实质超越。但掌握操作系统实现细节，躬身实践，或可为来日研发下一代操作系统奠基。

今日，量子计算机、虚拟现实、增强现实、人工智能、脑联网及意识解码等技术崭露头角。受制于基础科学水平，皆在瓶颈期。但可预见的未来基础科学必有突破，届时科技革命会再次降临。

我们正为之而努力！

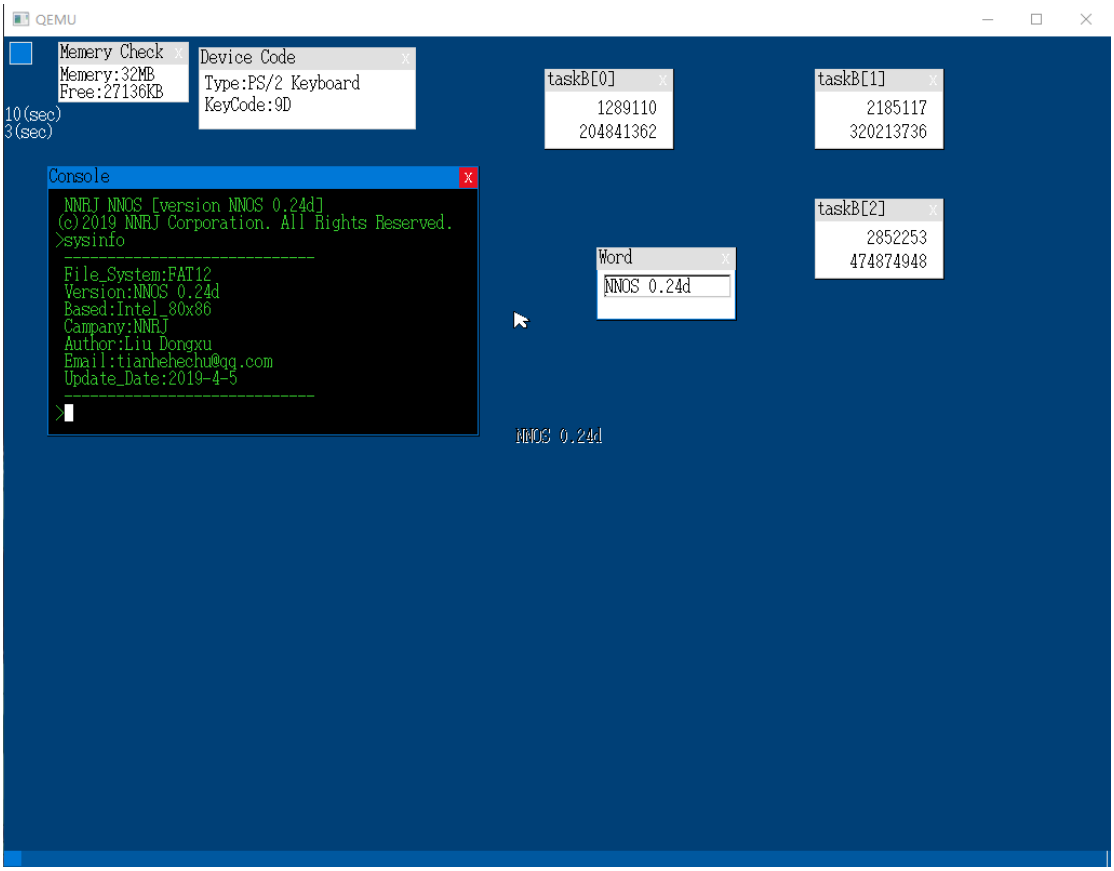


图 1 正在 QEMU 虚拟机中运行的 NNOS_0.24d 操作系统雏形

年年软件

2020 年 1 月 19 日

一 文档约定

1.1 约定

本文档使用「」代替“”作为引号。
本文档小节标题标红色者推荐重点阅读，标题标*灰色*者可略过不读。
本文档小节标题标注~~删除线~~者，为已过时（因系统更新等）的内容。
本文档小节标题标蓝色者推荐**重点阅读并动手实际操作。**
本文档内容加粗者，为该段叙述的关键词或术语。
本文档部分术语已设置百度百科链接，按 CTRL 键并单击即可访问。
本文档力求精简，涉及操作系统理论时只要一般计算机专业人员可以理解的，都不作额外说明，以将主要精力用于动手开发。

1.2 术语

在本文叙述中，使用「本系统」来代指「年年操作系统」。
本系统**尚未正式命名**，暂用惯用名 「年年」，系统内核中称“NNOS”。
「操作系统」并无统一明确定义，大可尽情发挥，但目前针对已有 OS 的理论框架已相当完备，且硬件架构和指令集有所限定，抛弃现有框架显得南辕北辙。故本系统仍然以经典四大模块为实现对象，并辅以图形界面。

1.3 快速运行指南

本指南供零基础者迅速搭建环境并运行 NNOS 发行版。(安全起见，暂不提供物理机运行方案。)
系统开发者请移步第五章开发环境搭建获取开发环境搭建的详细说明。
①下载镜像
[下载发行版]（百度网盘提取码：czex）
(因众所周知的原因，暂时无法提供可用的 github 下载链接)
（若上方链接失效，请加 QQ 群[757581301]，在群文件中下载）
②安装 VMware Workstation 虚拟机（其他虚拟机亦可）
③新建一个虚拟机，在选项卡中添加 NNOS 镜像
④开机运行
（详细步骤见发行版文件夹中的 **NNOS 快捷安装指南.gif**）

二 系统概述

本系统基于 Intel 80x86 架构。麻雀虽小，五脏俱全，具有基本的内存管理、I/O 管理、进程管理和文件管理系统，并提供较为友好的图形界面。

2.1 CPU 架构

本系统基于 Intel **80x86 系列架构**，支持 i386 及以上机器，32 位。

2.2 模块划分

本系统设计并开发实现的主要模块包括 IPL 引导程序、实模式模块（含过渡程序）、32 位保护模式系统主体、基本内存管理、基本 I/O 设备管理、基本进程管理、基本文件管理、基本 PIC 中断处理器、基本 PIT 间隔定时器、简单 TSS 多任务、基本点阵字库支持、VBE 显卡兼容、图形处理引擎、层叠窗口、扁平化图形界面、控制台、任务管理器、文件浏览器，并计划提供对 GB2312 中文字库解析引擎。

2.3 功能框架

本系统功能可直观表示系统功能图，如图 2.1.1 所示。

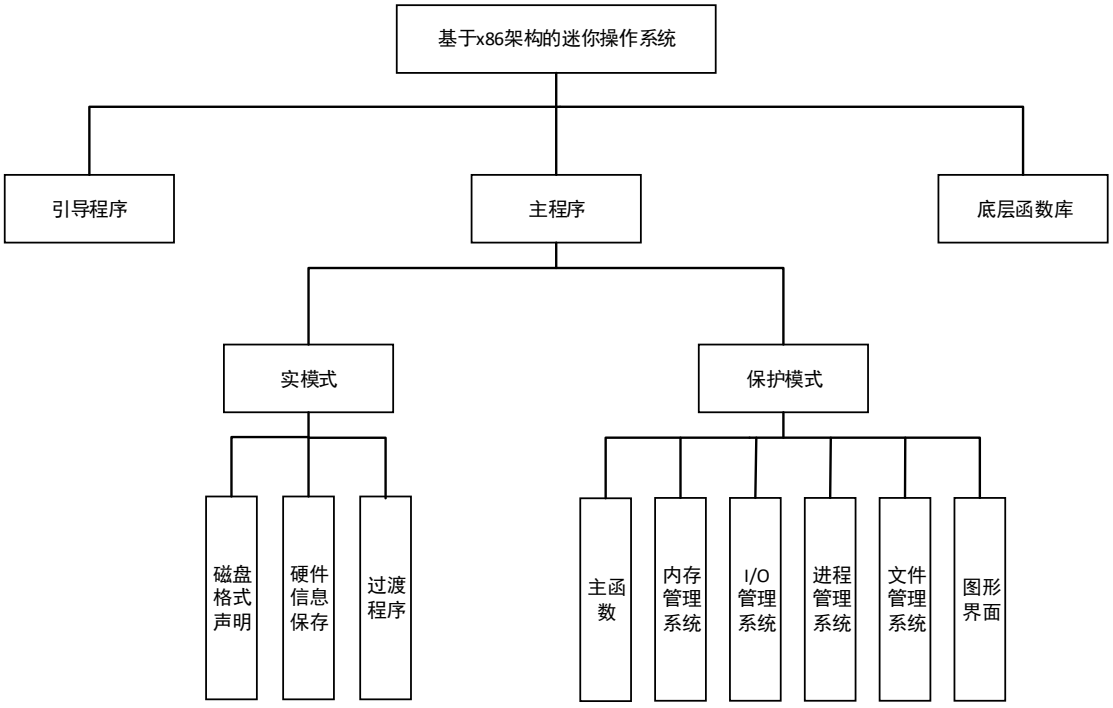


图 2.1.1 系统功能图

本系统实现操作系统的基本功能和图形界面。各个功能都有一个或多个层次、模块与之对应。对应关系见详细设计。

本系统整体功能可分为引导程序、**底层函数库**和**主程序**三部分。其中主程序又分实模式和保护模式两个部分。实模式部分有三个子功能，即磁盘格式声明、硬件信息保存和实模式到 32 位保护模式的过渡。保护模式分为主函数入口、内存管理、I/O 管理、进程管理、文件管理和图形界面处理六大系统性功能。

本节对主程序保护模式中的四大核心模块进行概述。其他功能与具体硬件或模块关系紧密，见详细设计章节。

2.3.1 内存管理系统概述

内存管理系统完成内存检查、分配与回收，用精心设计的数据结构和算法提高内存利用率。

其中最基础的两个子功能是空闲内存管理和内存分配。各种分配方法优劣的比较在此不作赘述。本项目算法简洁优先，目前空闲内存管理采用空闲表法，内存分配采用简单而不失高效的首次适应算法。详见详细设计。

2.3.2 I/O 管理系统概述

I/O（下称 I/O 设备、设备或 I/O）管理系统完成 I/O 中断处理、解析、端口分配。功能完备的操作系统，I/O 管理异常庞杂。因为连接在操作系统上的 I/O 设备种类、型号各异，不可预知。

本项目目前仅实现了对显示器、磁盘、鼠标、键盘等基本 I/O 的管理。若要支持所有主流 I/O 设备，一人之力无法完成，因为说服 I/O 厂商为一个尚未成形操作系统开发驱动程序几无可能。

2.3.3 进程管理系统概述

进程管理解决进程间资源共享问题。本系统目前已实现多任务，采用抢占式、时间片轮转算法，并设置优先级队列，同优先级进程优先执行在高优先级队列者。其他更复杂的进程管理将在今后版本中支持。详见详细设计。

2.3.4 文件管理系统概述

文件管理基于特定逻辑结构实现文件增删查改、打开与关闭、共享和保护。本系统目前仅支持简单的文件管理，以拓展名区分文件类型。支持文本文件查看，支持汇编语言与 C 语言编写的可执行文件的运行。详见详细设计。

2.4 层次框架

本系统各层次、各模块关系见如图 2.4.1 所示。

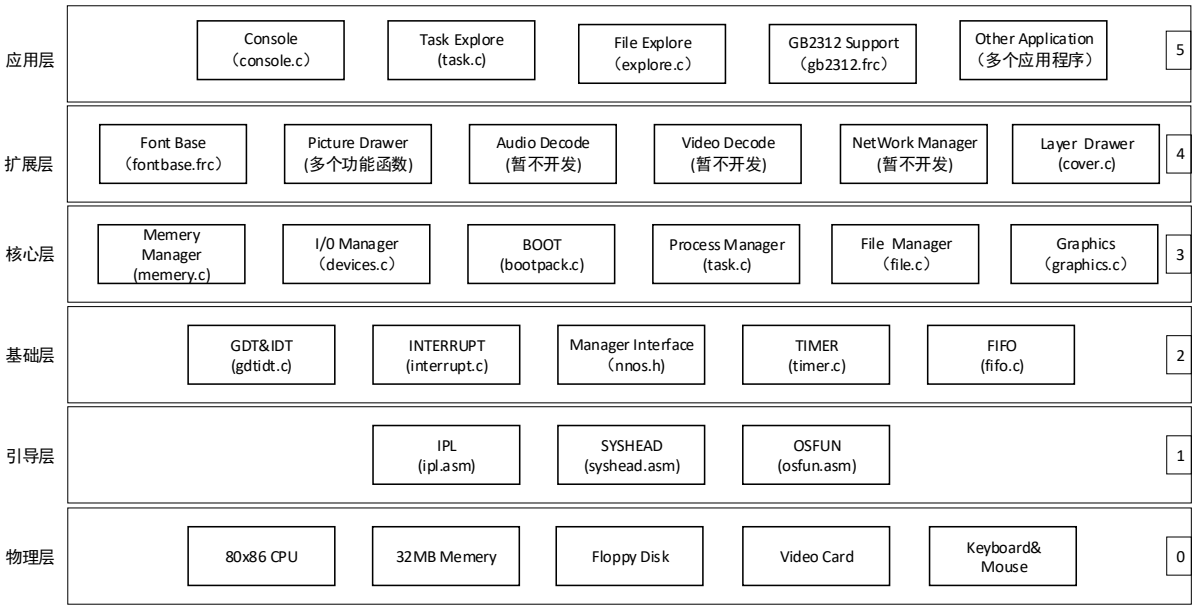


图 2.4.1 系统层次框图

本系统共分为五层，自下而上分别为引导层、基础层、核心层、扩展层和应用层。系统硬件和 BIOS 等最低硬件配置称第 0 层，不在本系统范围内，但在图中画出以便说明软硬件关系。BIOS 往上，依次编号为第 1-5 层。编号降序方向称“下”，编号升序方向称“上”。下一层为相邻的上一层提供服务，上一层通过接口调用下一层的服务完成自身功能。从引导层（包括引导层）到核心层，为操作系统的内核部分，共三层；核心层以上（不含核心层）为操作系统外壳，共两层。

左侧为各层名称，右侧标有本层编号，层内方框为各层模块，方框中标明了本模块名称，名称下括号中为本模块对应主要源文件名或暂时不开发的模块。

本文档只对软件模块加以说明，硬件指令、参数乃至格式声明大都是硬件设计者所作硬性规定，可在相应手册中查阅，不得不提及的部分只做简略说明。考虑到能力和成本，扩展层的 Audio Decode（音频解码引擎）、Video Decode（视频解码引擎）和 NetWork Manager（网络管理系统）三个模块本项目不实现。下面对本系统的各层进行详细介绍。

2.4.1 物理层

物理层在层次框图中为第 0 层，含运行必须的硬件及 BIOS，非本系统成分。包括 80x86 CPU（下称 x86，PIC[1]和 PIT[2]已集成，部分型号集成 Cache）、32MB 以上内存、基本 I/O 设备（磁盘、显示器、键盘、鼠标）。现对本系统所

1 PIC，Programmable Interrupt Controller（可编程中断控制器）缩写。又称 APIC。本文仍称 PIC。
2 PIT，Programmable Interval Timer（可编程间隔定时器）之英文首字母缩写。

涉硬件及其规定作概述。

（1）CPU 的实模式和保护模式。 本系统基于 x86，使用 x86 指令集。80x86 是 Intel 推出的以 8086 为鼻祖的一系列处理器的统称。8086 到 80286 为 16 位 CPU，386 以后为 32 位，皆采用 CISC、小端模式，向后兼容。本系统适用 386 以后的 x86 处理器。

x86 的段寄存器为、数据总线皆 16 位，地址总线 20 位。运行状态分为保护模式和实模式。内存划分为逻辑段，寻址方式为 “段：偏移”。实模式下段值意义为某地址起的一段内存，段最大 64KB（2^16）可寻址 1MB（2^20）内存。保护模式下，段值仍存 16 位寄存器，但意义为索引，指明表项。表项中定义起始地址、界限和属性。用于全局的表称 GDT（Global Descriptor Table，全局描述符/表），有且仅有一张，指明段信息供访存，GDTR 存放其入口地址。用于中断的表称 IDT（Interrupt Descriptor Table，中断描述符/表），记录中断号与调用函数间的， IDTR 存放其入口地址。，用于局部的表称 LDT（Local Descriptor Table，局部描述符/表），记录局部段信息。本系统仅实现 GDT 和 IDT。

设物理地址为 PHA，“段：偏移”用[ES:BX]表示，实模式下物理地址计算公式为：

$$PHA = ES \times 16 + BX \tag{1}$$

保护模式下，情况更复杂,涉及具体 GDT、LDT 及其选择子。切换到保护模式，需打开 A20 地址线（为兼容默认关闭），开启后 32 位保护模式最大支持 4GB 内存空间。

（2）寄存器。x86 有 8 个 16 位通用寄存器（四个可按高八位和低八位拆分使用），1 个指定地址寄存器，1 个标志寄存器，4 个段寄存器。通用寄存器中又有两个寄存器用于特殊用途。通用寄存器及其用途见附录Ⅰ表 9，专用寄存器及其用途见附录Ⅰ表 10。

（3）1.44M 标准软盘。本系统早期安装在 3.5 寸软盘中，它有 80 个柱面，80 个磁道（正反面各 80 个），每磁道 18 个扇区，每扇区为 512 字节，正反面各 1 磁头。总容量 1.44MB，故虚拟软盘中文件不足 1.44MB 也要写 0 凑齐。柱面、磁道、磁头从 0 编号，扇区从 1 编号。正面为 0 号磁头，反面为 1 号。0 号磁道在外，不同磁道周长不同但扇区数同（某些新型磁盘不同），越外磁密越低，数据越安全，故引导程序首址在 0 磁道。本文用 CHS（Cylinder-Heades-Sector，柱面-磁头扇区）法来表示软盘扇区地址。

本系统软盘文件格式为 FAT12（File Allocation Talbe，文件分配表）。实模式下读取软盘信息时需要使用 BIOS 指令，这些指令为硬性规定，见附录Ⅰ表 11。

（4）显卡和显卡模式。显卡用于存储画面信息，和内存一样可按址访问。它模式多样，本系统支持 VGA 显卡模式的 320×200×8 位彩色低分辨率模式和 VBE 显卡模式除 1280×1024×8 位彩色外的所有高分辨率模式。显卡模式切换在实模式下完成，规格见附录Ⅰ表 12、表 13。

2.4.2 引导层

引导层在层次框图中为第 1 层。主要用于引导系统启动、读取和保存硬件

信息、完成实模式到保护模式的切换、提供直接访问硬件的底层函数，用汇编语言编写。本层主要包括 IPL（Initial Program Loader，启动程序加载器）、系统实模式和系统底层函数库。

（1）IPL 引导程序（IPL）。用于引导 OS 启动。大小为 512 字节，存于软盘的 C0-H0-S1，此扇区称引导扇区。POST 通过后，此扇区被装入内存的 0x7c00h 并执行。须以 0xaa55[1]结束方能被识别为引导程序。此外，IPL 中还声明了 FAT 格式、并输出 OS 信息（系统启动极快，此信息不会被肉眼察觉，但在发生异常时可提供帮助）。IPL 详见详细设计。

（2）系统实模式（SYSHEAD）。汇编语言编写。IPL 引导成功后控制权交实模式。实模式可对硬件敏感操作（保护模式不可），如 BIOS 中断、读写硬件等。OS 最终在保护模式运行，所需必要信息（如显存地址、键盘指示灯初始状态、系统装载位置等信息）又须敏感操作。故切换前由实模式完成信息读取并存于内存的指定位置（见附录Ⅲ），保护模式只需要从相应位置读取相应即可。此外，实模式下还要完成硬件兼容检测（如显卡模式的选择）、硬件设定、从保护模式的过渡、临时 GDT 设定。详见详细设计。

（3）系统底层函数（OSFUN）。本系统主体用 C 语言编写，但底层硬件操作（如 I/O 端口读写）C 语言无法完成，须使用汇编写函数库供调用。涉及的操作有 CPU 停机、中断屏蔽、读写 I/O 端口、读写 FLAGS（保护现场）和中断响应等。详见详细设计。

引导层的三个模块，编译成目标文件与系统主体链接，从而为系统主体提供支持。

2.4.3 基础层

基础层主要包含 GDT/IDT、PIC、PIT 和 FIFO 相关的函数实现，为 OS 提供基础性支持。32 位保护模式入口（boot）包含于此层。本层是整个 OS 的地基。

（1）GDT 和 IDT 模块（GDT&IDT）。二者本质上都是数据结构，可用 C 语言结构体类型定义。GDT 需详细定义段起始地址、段界限、段属性、访问限等信息。IDT 中需定义段偏移、段选择子（段号）、访问权限等信息。二者都是用于分段存储管理。

GDT 以 CPU 信息为基础，受硬件规格制约。x86 要求段信息按 8 字节写存，故 GDT 须为 8 字节。32 位模式下最大寻址空间 4GB[2]，故段上限最大 4GB。段寄存器 16 位，用段号将各段索引，占 4 字节，低 3 位不可用，故段上限只能用 20 位。段访问权 12 位，高四位拓展访问权在段上限高位，低八位在段访问权限中。

IDT 用于中断处理，设备产生中断后，CPU 根据 IDT 中注册的函数完成中断处理。

（2）中断处理器模块（Interrupt）。完成 PIC 初始化、保护现场、中断处理。与中断相关的函数（无敏感操作）都整合在此模块，敏感操作调用 OSFUN

---

1 此十六进制数按照小端模式在内存中为 10101010 01010101，作为引导分区的引导标识。

2 标志位默认 0，段上限单位释为字节。置 1 时释为页，4KB。ES:BX 访存 1M，解释变化后为 4KB×1M＝4GB。

之服务。本模块依赖 PIC，现对 PIC 的工作机制作简要说明。PIC 间、PIC 与 CPU 间关系示意如图 4 所示。

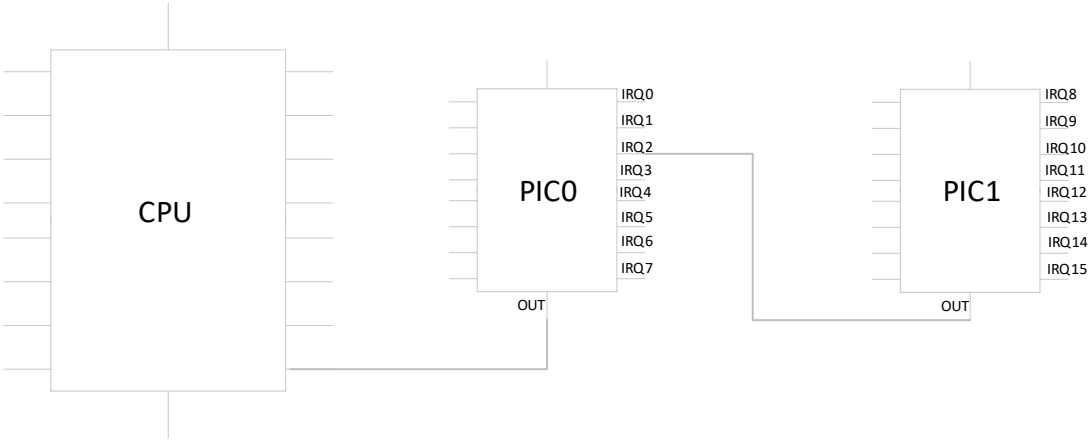


图 2.4.2　CPU 与 PIC、PIC0 与 PIC1 连接示意图

现代 CPU 已将 PIC 集成[9]。PIC 分主 PIC（称 PIC0）和从 PIC（称 PIC1），PIC0 与 CPU 直连，PIC1 通过 PIC0 的 IRQ2（Interrupt Reques 之缩写，硬件上代指对应管脚）与 PIIC0 连接，经 PIC0 与 CPU 通信。PIC 共 16 组输入管脚（主从各 8，自 PIC0 起编为 IRQ0~IRQ15），一组输出管脚（在 PIC0）。输入管脚中，除 IRQ2 外，剩余 15 个管脚用于中断处理（PIC0 负责 0 到 7 号中断， PIC1 负责 8 到 15 号）。PIC 时刻监测管脚传来的 IRQ 信号，收到信号后，若当前为中断允许状态，则将中断信号从唯一输出管脚输出，交给 CPU 处理。根据规定[10]，PIC 内部的寄存器通过指定的端口号访问，其中 IMR（Interrupt Mask Register，中断屏蔽寄存器,8 位）用于接收中断屏蔽操作数，ICW（Initial Contrl Word，初始化控制数据，8 位）用于初始化。端口间或外界向端口传送的操作数为 2 字节数据，通过 I/O 数据线传输，信号意义在硬件设计时硬性规定，不可更改。0xcd 用于 BIOS 中断，0x00~0x1f（此信号段用于 CPU 中断保护，防止应用程序越权）不能用于 IRQ。本系统使用 0x20~0x2f 接收中断信号，通过中断处理函数对 PIC 进行编程控制，实现中断管理。此外，部分机型开机自动产生 IR7 中断，须处理，否则会启动失败。

（3）间隔定时器模块（TIMER）。间隔定时器模块完成 PIT 初始化（实现时放在核心层 I/O 模块）、定时器分配、定时器释放、定时器设定、时刻调整、定时器测试等功能。硬件层面，PIT 每隔一段极短的固定时间间隔产生一次 PIT 中断，通过=编程=可以实现较精确的定时器，为系统的其他功能（如钟表、时间戳、调度算法、UI 动画等）提供支持。PIT 中断频率与 CPU 主频、用户设定的中断参数，中断频率为 ω，设主频为 f，设参数为 τ，则数值上，PIT 中断频率可通过以下公式求得：

$$\omega = \frac{f}{\tau} \qquad (2)$$

基础层的 TIMER 模块，对 PIT 进行了编程，满足定时器功能之需。

（4）缓冲区模块（FIFO）。本模块为系统慢设备（键盘、鼠标、打印机等）在内存开辟受保护的数据缓区域，保证设备数据安全，解放 CPU。本系统采用 32 位 FIFO（First IN First OUT，先进先出）缓冲区，为键盘、鼠标和定时器所共用，并对键盘和鼠标中断信号（信号标识数据来源和中断信息）进行了偏移。减少了资源占用、简化了缓冲区数据处理。本系统对缓冲区信号作

出出规定，信号范围与来源对应关系如表 2.4.1 所示。系统处理缓冲区数据时，先根据信号范围判断来源，再偏移为原数值作解析。

表 2.4.1 中断数据信号范围与设备中断来源规定

信号范围	设备中断信号来源	偏移	备注
0~1	光标定时器	0	
2~255	自定义	/	
256~511	键盘码	256	
512~767	鼠标输入	512	
768 及以上	自定义	/	

（5）管理接口模块(Manager Interface)。本模块将系统内核级别函数封装，包括所有函数声明和 API，为本系统其他模块及应用程序提供系统函数调用。接口存在权限管理机制，将内核与应用隔离，以防应用程序越权，保证系统安全。函数声明又包含了系统公用常量和公用结构体。

常量包括系统属性、BOOT 信息首地址、GDT 相关、IDT 相关、显卡常量、PIC 相关、PIT 相关、内存相关（内存描述和内存处理）、鼠标常量（鼠标硬件和鼠标指针）、键盘端口常量、FIFO 缓冲区（下称 FIFO）相关、桌面常量（引入 BOOT 信息后废弃）、图层常量、系统默认窗口定义、系颜色色号注册表（下称颜色表）、系统颜色注册表助记符（下称颜色名）、系统颜色注册表助记符别名（下称颜色别名）、字体常量、特殊地址常量、鼠标指针图像点阵（下称鼠标点阵）、键盘字符映射表（下称字符映射表）及其他常量。详见详细设计。

结构体主要有 GDT、IDT、TSS、BOOT 信息、屏幕、鼠标指针、内存空闲块、内存空闲块表、计时器、计时器列表、像素点、色块、图形、窗口、图层、图层列表任务、任务列表、控制台等等。页数限制，不再一一罗列，见详细设计。

2.4.4 核心层

核心层包含 OS 的四大核心模块，即内存管理、I/O 设备管理、进程管理和文件管理。系统图形界面包含在此层。四大模块是操作系统之根本[11]，引导层、基础层建设完成后，核心层为重中之重，优先于其他一切层次和模块。本层实现依赖基础层提供的服务。

（1）系统主函数入口（BOOT）。类似 C 语言的 Main 函数，实模式切保护模式后，从此处开始执行。BOOT 调用各模块服务，完成 OS 基本功能。执行到此，OS 才真正启动成功。习惯上系统的主体及其所在模块被命名为 BOOT，本系统沿用此名称。

（2）内存管理模块（Memery Manager）。本系统采用动态连续空间分配，空闲空间管理采用空闲表法，分配采用首次适应算法[12]。将内存按地址动态划分为若干块，程序申请内存时，调用本模块内存分配服务，获得所需空间。空闲表中，空闲块按容量从小到大有序排列，分配前对请求者所申请容量 4K 舍入以尽可能减少外部碎片。当某块被释放时，涉及到内存块增减合并，有多种

情况，相对复杂，现以图 2.4.3 示意，简要说明。

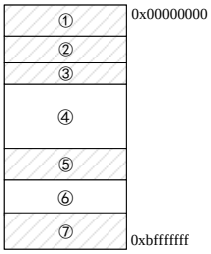


图 2.4.3 某时刻内存分配状态示意图

图中外层矩形代表从 0x00000000~0xbfffffff 的内存空间。阴影部分为已分配内存块，空白部分为空闲内存块。为方便说明，内存块从上到下按地址递增编号，号码标于图中。图示内存块从小到大依次为③②①⑤⑥⑦④，其中①与⑤、⑥与⑦大小相同。在空闲块表中，空闲块表项以⑥④的顺序排列，当某块释放时，空闲块表项先按地址重排为④⑥。

已分配内存块释放时，空闲块表变化的部分不同情况如表 2.4.2 所示。

合并前，为找出连续的块，需对空闲块表按地址顺序重排。合并后，为不影响后续分配，须再按大小排序。实际实现时，还涉及具体数据结构，情况更加复杂。前无空块或后无空块时，需对首尾地址作特殊处理。不同数据结构，排序、移块操作各不相同。详见详细设计。支持最大 32MB 内存，起始 4MB 预留给操作系统使用。

表 2.4.2 某时刻某块释放时内存空闲块表可能的变化情况

释放的块	空闲（释放前）	情形判断	按址排序（含待释）	空闲（释放后）	处理/表项增减
②	⑥④	前后无空块	②④⑥	②⑥④	无合并，加 1
③	②⑥④	前后有空块	[②③④]⑥	⑥[②③④]	②③④并，减 1
⑦	⑥[②③④]	后无（尾）	[②③④][⑥⑦]	[⑥⑦][②③④]	⑥⑦并，不变
①	②⑥④	前无（首）	[①②]④⑥	[①②]⑥④	①②并，不变
②	①⑥④	后无空块	[①②]④⑥	[①②]⑥④	①②并，不变
③	⑥④	前无空块	[③④]⑥	⑥[③④]	③④并，不变

[注]本表中，序号使用中括号"[]"括起来表示连续的序号合并为一个整体。

（3）I/O 管理模块（I/O Manager）。I/O 管理模块目前支持磁盘、键盘、鼠标和显示器的管理。本模块用于为 I/O 设备提供中断响应服务、读取和解析中断信息并写入 FIFO。本模块依赖基础层 INTERRUPT 和 OSFUN 模块之服务，对 I/O 端口进行编程，并依赖基础层 FIFO 创建缓冲区，将键盘、鼠标中断数据存入 FIFO，供 BOOT 使用。本模块提供键盘、鼠标缓冲区数据解码器，从而实现鼠标指针定位、移动及键盘按键信息的解析。

现结合硬件对本系统键盘鼠标 I/O 中断处理过程作出简要说明。

CPU 速度极快，I/O 则相对极慢。为避免 CPU 徒劳等待 I/O，诞生了各种 I/O 处理机制[13]。本系统采用较为简单的 I/O 中断响应机制。

键盘和鼠标是典型输入设备。键盘控制电路与 PIC0 相连，管脚 IRQ1。处理键盘中断前，键盘控制电路（下称 KCMD）须初始化，初始化前须确保 KCMD 电压稳定。初始化主要设定 KCMD 的模式（键盘模式、鼠标模式）和 FIFO。处理键盘中断时，先向 PIC0 的 IRQ12 发送指令 0x61，告知 PIC 本次中断已受理，数据随后被被写入 FIFO。

鼠标应用较晚，连接在 PIC1 上，和键盘一样需通过 KCMD 完成中断。中断信号发送到 PIC1 的 IRQ12 端口，紧接着传给 PIC0 的 IRQ02，然后提交到 CPU 作出响应。

（4）进程管理模块（Process Manager）。本模块含任务管理和进程管理。系统资源有限，运行多个进程时，操作系统须对资源进行合理分配、回收。CPU 同一时刻只能执行一条指令[14]。多个任务同时运行时，任务被标识不同的优先级。系统维护多个优先级队列，根据任务优先级、优先级队列选取当前要运行的任务。时钟周期被分割为碎片，系统在多个任务间切换，使之轮流占有 CPU，切换时间极短，看起来多个任务同时运行。系统为进程创建 PCB，完成进程的创建、资源分配和销毁。

（5）文件管理模块（File Manager）。FAT12 文件系统存在一个文件分配表，称 FAT，为整个磁盘文件的索引，该索引标识了文件内容起始地址、文件名、文件拓展名、文件权限等信息。本系统通过读写 FAT，结合 OSFUN 之服务，完成文件的增删查改、权限设置和可执行文件运行。

（6）图形处理模块（Graphics）。本模块整合本系统所有图形渲染引擎（Engine，本质为函数或组件）。包括像素渲染、色块渲染、增强色块渲染、图形绘制、窗口绘制、字体渲染、鼠标指针解析、UI 初始化、快速窗口绘制等。根据 BOOT 传参（图层、显卡首址或待绘图形）在显存或相应图层写入像素颜色数据，依赖公用常量中的颜色表和绘图相关结构体（像素、色块、字体、窗口、鼠标指针等），实现文字、图片、窗口的显示。

引导层、基础层和核心层是内核部分，本系统内核部分的概要设计阐述完毕。

2.4.5 扩展层

自本层向上为系统外壳。本层为 OS 的必要扩展，包括字库、网络管理和层叠窗口处理。基本字库模块用于支持字符显示。出于实际需要，本模块的字体和字体解析引擎同字体渲染引擎一起整合在图形处理模块。本系统目前仅支持点阵字库。

字体库采用文献推荐的 hankaku，该字体库的作者是平木敬太郎先生和 Kiyoto 先生，川合秀实先生得到授权并授权读者自由使用。本系统把该字体库重命名为 fontbase，并对点阵的标识做出适应性调整，但核心仍是 hankaku，在此对三位先生表示感谢。

（1）基本字库模块。与字体库相适应，支持 16×8 位点阵字体和 ASCII。后期将支持 GB2312 标准中文字体库。安全考虑，GB2312 等其他字体库皆置于

应用层。

（2）图层处理模块（Layer Drawer）。本模块用于图形界面的图层分配、释放、设定、刷新和移动。显示在屏幕上的每个窗体（包括桌面、指针和光标）都存储于各自缓冲区，此缓冲区之指针及缓冲区属性构成的结构体称图层。引入图层的目的在于实现各窗体的自由移动、开关，防止窗体移动、刷新后覆盖掉其他窗体。

（3）图像处理引擎。用于图片解析，支持主流格式。非核心，低开发优先级。

2.4.6 应用层

应用层包含面向用户的基础应用程序，如控制台、任务管理器和文件管理器，非 OS 的核心成分。控制台是系统面向用户的接口，用户可通过控制台命令来完成人机交互。任务管理器是进程管理信息的图形化展现，提供给用户有关系统正在运行的进程的参考信息。文件管理器是文件管理系统的图形化，使用户能用键盘和鼠标操作文件的增删改查。应用层其他应用，皆非项目重心，视开发进度增减。若时间充裕，计划开发时钟、计算器、画图、文本文档、图片查看器、扫雷游戏、日历及适用本系统的编译器。此外还有图形界面的 CSS3 标准化，即支持通过 CSS3 自定义图形界面。

各层次、模块调用细节见详细设计。至此，本系统五个层次全部介绍完毕。

2.5 整体流程

系统整体运行流程如图 2.5.1 所示。

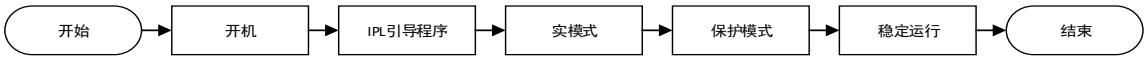


图 2.5.1 系统整体运行流程

计算机 POST 时，任何核心硬件故障都不能启动，更不能运行 OS。POST 通过，BIOS 找到启动盘，读取并加载 IPL。IPL 完成必要准备，控制权交给 OS。OS 起初在实模式运行，完成保护模式无权的操作、存储硬件信息后，进入过渡程序。过渡程序将 OS 从实模式切换到保护模式，保护模式下，OS 完成最后准备工作，直至启动成功或失败。启动成功后，系统稳定运行直至关机或断电。系统运行的详细流程见详细设计。

2.5.1 IPL 运行流程

IPL 的整体运行流程如图 2.5.2 所示。

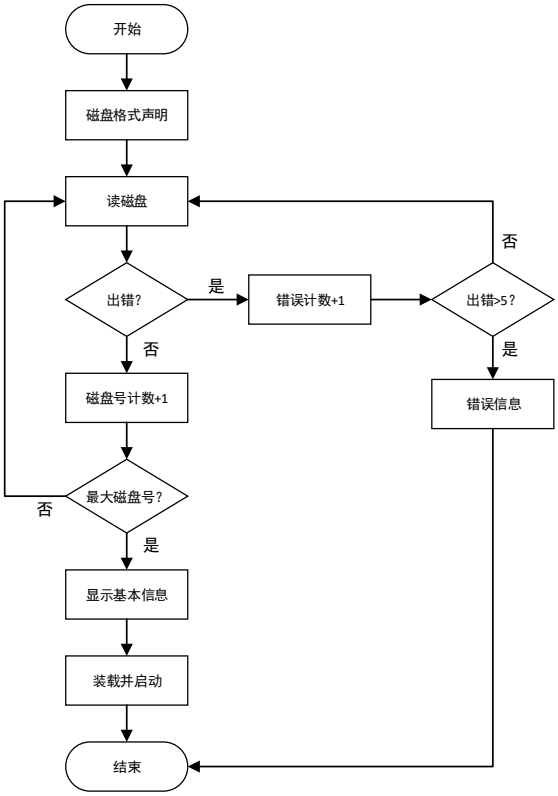


图 2.5.2 IPL 整体流程图

IPL 按 FAT12 文件系统标准声明磁盘格式，然后读磁盘的到设定的最大柱面号。若读盘出错，则错误计数加 1，累计出错 5 次视为不可读，打印错误信息结束运行，启动失败。若读盘顺利，则打印 OS 信息（架构、版本号、联系方式等），随后装载系统实模式，控制器移交给 OS，IPL 完成引导任务。详见详细设计。

2.5.2 系统实模式运行流程

本系统实模式整体运行流程如图 2.5.3 所示。

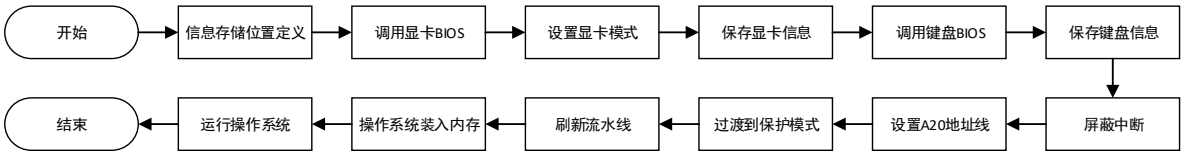


图 8 系统实模式整体流程图

IPL 引导系统启动后，控制权交给系统实模式，实模式中定义了系统关键信息存储位置。首先判断并设定显卡模式（VGA 和 VBE，可细分，详略），设置完毕后将显卡信息和键盘 LED 状态等存到内存指定位置。然后屏蔽 CPU 级别中断，开 A20 地址线以切换到保护模式。切换后处理器对指令的解释发生变化，需立刻刷新流水线。刷新完成后，程序将 BOOT 装入内存，控制全交给保护模式。

2.5.3 保护模式运行流程

保护模式的整体流程如图 9 所示。实模式切到保护模式后，BOOT 获得控制权。首先初始化 GDT 和 IDT，完成其属性设定；然后初始化 PIC 和 PIT，完成 PIC 设定。之后初始化 FIFO 和已设定的定时器。然后开中断（实模式向保护模式过渡前屏蔽了中断），初始化 KCMD（此时键鼠尚不可用）；然后初始化画板（注册系统颜色表）、初始化屏幕、桌面和鼠标图形（此时桌面和鼠标指针被绘制出来）；然后激活鼠标（此时可响应键鼠可用）；然后初始化内存管理、进程管理、文件管理系统，执行内存检查、获取内存信息，将信息在图形窗口展示；初始化 TSS，读取 FIFO 数据，响应用户操作。

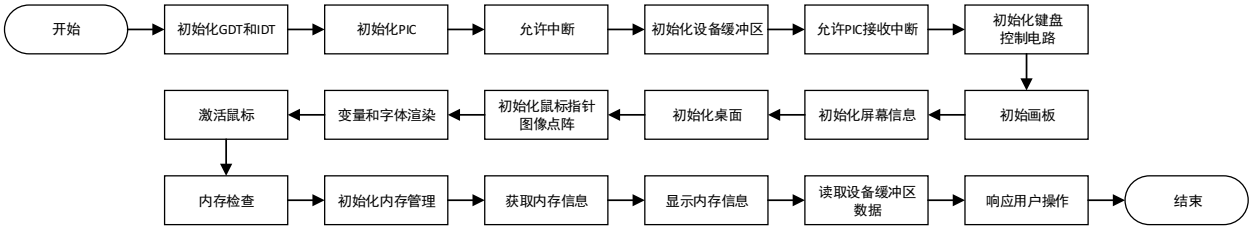


图 9 系统主体整体运行流程图

三 详细参数

本节结合具体实现，对本系统各层次、各模块、各函数实现进行详实说明。

3.1 项目源文件结构

本项目源文件目录如图 3.1.1—图 3.1.4 所示。其中图 3.1.1 和图 3.1.2 为同一目录，为保证展示效果，截为两段。为减少篇幅，已将中间文件删除。

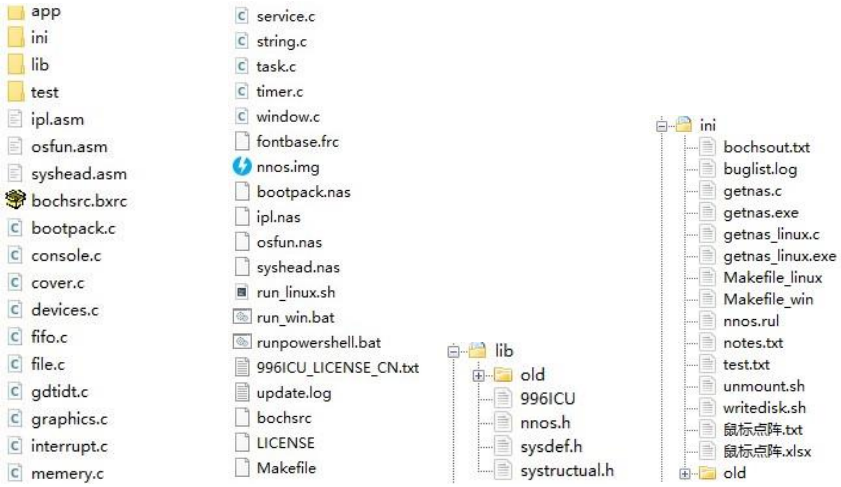


图 3.1.1(左 1)、图 3.1.2(左 2) 项目源文件主目录、图 3.1.3 项目源文件 lib 目录（左 3)、

图 3.1.4 项目源文件 ini 目录（右 1）

主目录包含汇编和 C 语言源文件，编译脚本，系统配置 (ini)和系统依赖(lib)目录。其中同名文件拓展名分别为.asm 和.nas 的功能相同，具体实现略异。.nas 文件由转换程序从.asm 转换而来，实现对 nask 的兼容。上图重要源文件简介见附录Ⅱ表 15。

3.2 系统常量定义

系统常量指系统运行过程中数值不会变化的量，使用 C 语言预处理实现。

3.2.1 宏定义

本小节将以表格形式对本系统所有常量作说明（表格过多，页数限制，正文部分不再展示，见附录Ⅲ表 16~37。具体实现的文件为 syshead.h。

3.2.2 静态数组

系统颜色表、鼠标指针点图像和字符映射表使用静态数组定义。

（1）系统颜色信息注册表。红绿蓝为光的三原色，三者以不同的比例混合可形成其他颜色。与之对应， RGB 表示法（下称 RGB）用红绿蓝不同值混合表示复合色。8 位色彩显卡用 8bit 表示一个颜色值，范围为 0~255。本系统调色板采用 RGB，用 24bit 表示复合色。颜色表定义为静态字符型一维数组（加快解析速度）存储颜色的十六进制值，解析时三个一组，写入显存，实现颜色显示。使用时可直接指明颜色序号，为方便识记在常量中定义了颜色名和颜色别名指代相应颜色号。

类型：static unsigned char；数组名：table_rgb[]；数组大小：COLORNUM * 3；维度：一维；成员，见附录Ⅳ表 38，系统颜色信息注册表。。

(2)系统鼠标指针点阵。鼠标指针图像为 16×16 像素，其形状、颜色分布用静态二维数组定义，使用 graphics.c 的鼠标指针图像解析引擎对数组进行解析，即可在绘制出鼠标指针图像。渲染后的点阵数组如图 14 所示。

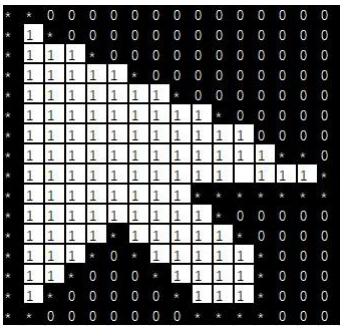


图 14 鼠标指针点阵图像映射表

数组类型：static char；数组名：CURSOR_GRAPH；数组大 CURSOR_WIDTH，CURSOR_HEIGHT；维度：二维成员，见图 14。为便查看，对上

图作了渲染。每行为一个一维数组，各一维数作为二维数组成员。图中“0”处被解析为背景色，“1”处被解析为 COL8_FFFFFFFE，“*”处被解析为 COL8_00000000。解析引擎实现见下文。

（3）键盘字符映射表。用户按下按键后会产生键盘中断，中断信息为相应键码（分为键按下和弹起两种，按下值与 ASCII 通），键码被中断处理程序写入 FIFO，解析程序取出减去偏移量（KEY_DATA_BASE），查找本映射表得相应字符，送到字体绘制引擎，字体绘制引擎调用绘制函数和字体库将字符在的像素信息写入显存或图层在屏幕上显示。数组类型：static char；数组名：KEY_TABLE；数组大小：KEY_TABLE_SIZE；数组维度：一维；成员：与 ASCII 表编号对应，见附录Ⅴ表 39。

3.2.3 专用类型定义

特殊位置数据类型具有特殊意义，用 typedef 将 C 语言默认数据类型预定义，在特殊位置定义此类型变量时使用此预定义名称代替。这样做的代价是编译器无法智能地对该变量初始化，必须手动初始化，否则会编译时会产生警告。

```
/*专用类型定义*/
typedef int COUNT; //定义 COUNT 类型（实质为 int 的别名，计数时使用，使之易于理解）
```

3.3 系统结构体定义

本系统数据结构定义使用 C 语言结构体实现[15]。32 位平台中 char 占 1 个节，short 占 2 字节，in 占 4 字节，定义结构体时须区分。

3.3.1 GDT 结构体和 IDT 结构体

GDT 和 IDT 是 OS 保护模式最重要的数据结构。GDT 用于分段，IDT 用于中断。保护模式按[ES:BX]访存，概要设计已阐述。GDT 用 8 字节描述段基址、上限和属性（权限）等，供给 CPU 访存使用。GDT 本身存于内存，长度可变，最大为 8K（16 位段选择子，13 位作为索引，剩余 3 位用作寻址，故最大 8K）。

段选择器中存段选择子，为访存时 GDT 索引号。段选择子 16bit，各位有特定含义，详见源文件注释。本系统 GDT 和 IDT 结构体 UML 图如图 15 和 16 所示。

图 15 GDT 结构体 UML 图（左）、图 16 IDT 结构体 UML 图（右）

GDT 结构体的定义及必要注释如下：

```
/*=全局描述符(GDT)*/
typedef struct SEGMENT_DESCRIPTOR{ //以 CPU 信息为基础的结构体，存放 8 字节内
容
    short LOW_LIMIT;                //低位上限
    short LOW_BASE;                 //低位基址（低位 2 字节基地址）
    char MID_BASE;                  //中位基址（中位 1 字节基地址）
    char ACCESS_PER;                //操作权限（禁止写入、禁止执行、系统专
用）
    char HIGH_LIMIT;                //高位上限
    char HIGH_BASE;                 //高位基址（低、中、高 32 位,三段，兼容
80286）
}SEGMENT_DESCRIPTOR;
```

IDT 结构体定义及必要注释如下：

```
//=====中断描述符(IDT)=================================================
typedef struct GATE_DESCRIPTOR{ //以 CPU 信息为基础的结构体，存放 8 字节内容
    short LOW_OFFSET;               //低位偏移
    short SELECTOR;               //段选择子（段号）
    char DW_COUNT;                  //计数器
    char ACCESS_PER;                //段访问权限
    short HIGH_OFFSET;              //高位偏移
}GATE_DESCRIPTOR;
```

3.3.2 FIFO 缓冲区结构体

FIFO 缓冲区结构体 UML 图 17、图 18 所示

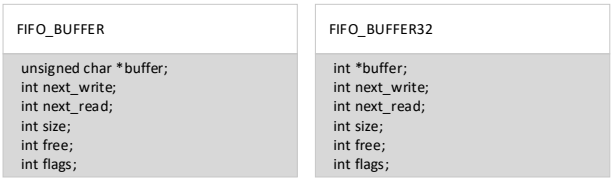


图 17 FIFO 缓冲区 UML 图（左）、图 18 32 位通用 FIFO 缓冲区 UML 图（右）

键鼠专用字符 FIFO 结构体定义（左）、32 位通用 FIFO 结构体定义（右）：

```
typedef struct FIFO_BUFFER{
    unsigned char *buffer; //缓冲区首地址
    int next_write; //写指针
    int next_read; //读指针
    int size; //总容量
    int free; //空闲容量
    int flags; //标志位
}FIFO_BUFFER;
```

```
typedef struct FIFO_BUFFER32{
    int *buffer; //缓冲区首地址
    int next_write; //写指针
    int next_read; //读指针
    int size; //总容量
    int free; //空闲容量
    int flags; //标志位
}FIFO_BUFFER32;
```

3.3.3 BOOT 信息结构体

UML 图如图 19 所示。

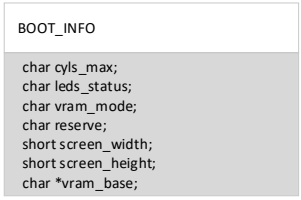


图 19 BOOT 信息结构体 UML 图

BOOT 信息结构体具体实现及必要注释：

```
/*BOOT 信息结构体*/          //信息数据来自于 syshead.asm
 typedef struct BOOT_INFO{   //指针指向相应信息
    char cyls_max;           //磁盘最大装载柱面号（启动区读硬盘读到何处为止）
    char leds_status;        //键盘 LED 指示灯状态
    char vram_mode;           //显卡模式（位数，颜色模式）
     char reserve;          //无定义，保留区域
    short screen_width;      //显示器行分辨率（单位：像素）
    short screen_height;     //显示器列分辨率（单位：像素）
    char *vram_base;         //显存首地址
}BOOT_INFO;
```

3.3.4 鼠标指针结构体

UML 图如图 20 所示

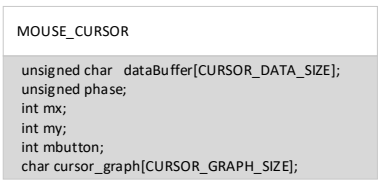


图 20 鼠标指针结构体 UML 图

自上而下（下同），成员依次为数据缓冲区、数据游标、横坐标、纵坐标、按钮类型、指针点阵图像数组。缓冲区和数组大小，皆由 syshead.h 中的常量决定（下同）。数据游标标识当前已接收的鼠标数据，解析引擎按三个一组解析，不足则等待或舍弃，详见鼠标解析引擎详细设计。

3.3.5 内存相关结构体

内存块结构体 UML 图 21（左）、内存空闲块表结构体 UML 图 22（右）：

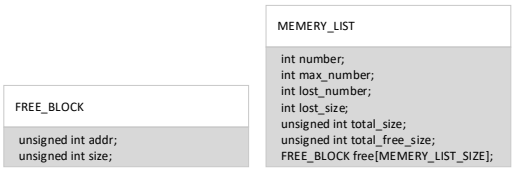


图 21 内存块结构体 UML 图（左）、图 22 内存空闲块表结构体 UML 图（右）

内存块体成员有内存块首地址、内存块大小。内存空闲块表成员有空闲块数、最大空闲块数、释放失败块数、释放失败次数、有效总量、空闲大小、表主体（存块）。

3.3.6 定时器相关结构体

定时器结构体 UML 图 23（左）、定时器列表结构体图 24（右）：

定时器结构体成员有结点指针（指向下一个定时器）、超时时间、中断信号、FIFO 指针和状态标志。定时器列表结构体成员有计数器、超时时间、运行中的定时器链表头指针、定时器列表数组。

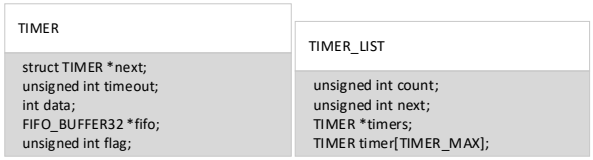


图 23 定时器结构体 UML 图（左）、图 24 定时器列表结构体 UML 图（右）

3.3.7 图形相关结构体

（1）像素点结构体 UML 图 25（左）、色块结构体 UML 图 26（右）：

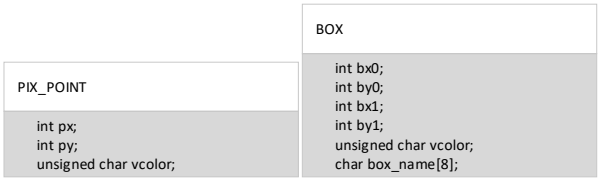


图 25 像素点结构体 UML 图（左）、图 26 色块结构体 UML 图（右）

像素点结构体成员：像素点横坐标、像素点纵坐标、无符号字符型颜色号。

色块结构体成员：横首、纵首、横尾、纵尾、色号、标识（限 8 字节）。

(2)图片结构体 UML 图 27（左）、窗口结构体 UML 图 28（右）：

图片结构体成员变量：横坐标、纵坐标、宽度、高度、数据数组基地址。

窗口结构体成员变量：类型（定义见宏）、横坐标、纵坐标、宽度、高度、标题指针、前景色、背景色、可见性（1 可见，0 不可见）、响应性（是 0 响应，1 不响应）。

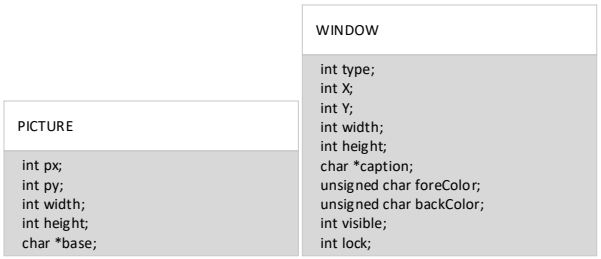


图 27 图片结构体 UML 图（左）、28 窗口结构体 UML 图（右）

3.3.8 图层相关结构体

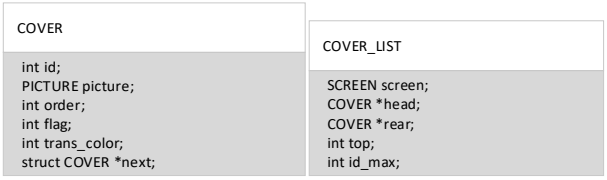


图 29 图层结构体 UML 图（左）、图 30 图层列表结构体 UML 图（右）

图层结构体 UML 图 29、图层列表结构体 UML 图 30：

图层结构体成员： ID、图片、层级、状态标志、透明色号、图层结点的指针。

图层列表结构体成员：屏幕信息、链头、链尾、顶部层级、最大层级。

3.3.9 任务相关结构体

（1）任务结构体 UML 图 31（左）、任务列表结构体 UML 图 32（右）：

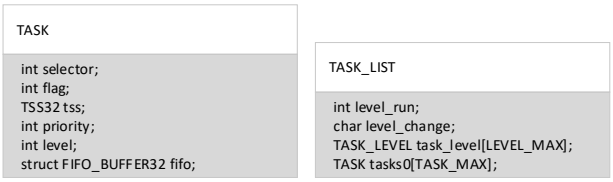


图 31 任务结构体 UML 图（左）、图 32 任务列表结构体 UML 图（右）

任务结构体成员：段选择子、状态标志、任务状态段、优先级、优先级队列、FIFO。任务列表结构体成员：运行中任务数、任务变化、运行队列、任务列表主体。

（2）任务优先级队列结构体 UML 图 33（左）、控制台结构体 34（右）：

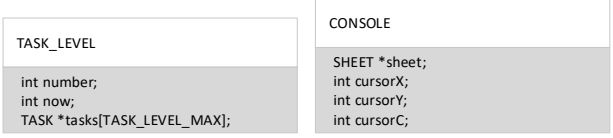


图 33 任务优先级队列 UML 图（左）、图 34 控制台结构体 UML 图（右）

任务优先级队列结构体成员：任务数量、正在运行的任务、优先级列表指

针。

控制台结构体成员：图层指针、光标横坐标、光标纵坐标、光标颜色号。

3.3.10 文件相关结构体

文件信息结构体 UML 图 35：

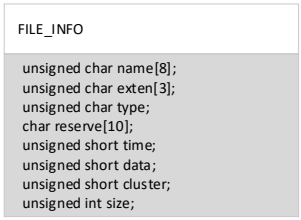


图 35 文件信息结构体 UML 图

文件信息结构体成员：文件名、拓展名、类型、保留区域、修改时间、修改日期、簇号、大小。

至此全部结构体之详细设计完成。

3.4 引导层详细设计

引导层包括 IPL、SYSHEAD 和 OSFUN，对应 ipl.asm/.nas、syshead.asm/.nas 和 osfun.asm/.nas 三个源文件。本系统在实现期间已经为代码加了大量详细注释，结合注释与代码，可以非常清楚地明白程序的具体实现。因论文页数限制，无法将添加着色的代码块，也使用过多文字描述。本节仅对部分模块作出说明，详见代码和注释。

3.4.1 IPL 详细设计

IPL 用于引导操作系统启动，整体流程见概要设计。本节对其具体实现作介绍。按源码上下文顺序进行，与程序际执行顺序不同。

（1）定义最大柱面号和装载地址。程序开头首先定义了最大磁盘柱面号。本系统 IPL 默认最大柱面号为 C10。

```
CYLS_MAX EQU 10  ;定义最大装载柱面号常数 CYLS_MAX,最大装载到第 10 柱面
ORG 0x7c00 ;指明程序装载地址。#主引导扇区拷贝到内存 0x7c00(与软盘拷贝位置一致)。
```

（2）磁盘格式声明。本系统采用 FAT12，读盘前须对磁盘格式作出严格声明。磁盘格式声明如下：

```
;FAT12 文件系统，IPL(512B,C0-H0-S1)、
;FAT 1(4068B,0-0-2~0-0-10)，FAT 2(0-0-11~1-0-1)，根目录、文件数据区
;描述用于标准 FAT12 格式的软盘(FAT12 格式软盘标准引导代码)
    JMP entry ;跳转到程序核心入口
    DB  0x90            ;[0]跳转指令（相当于"JMP 0x4e”）
```

```
    DB  "NNOS_IPL"      ;[3]引导区名称（可自定义，须为 8 字节，不满 8 字节用空
格填充）
    DW  512             ;[11]扇区的大小（字节， 512 字节。逻辑扇区经转换可得
CHS）
    DB  1               ;[13]簇大小(单位扇区，默认 1，FAT12 只理 4096 簇，最大
2M）
    DW  1               ;[14]FAT 起始位置(一般 1 扇区始）
    DB  2               ;[16]FAT 表记录数(默认 2，两表内容同， FAT2 为备份）
    DW  224             ;[17]根目录大小（根目录下目录/文件最大数量，默认
224。）
    DW  2880            ;[19]逻辑扇区总数（即磁盘大小，默认为 2880<0x0b40>个
扇区）
    DB  0xf0            ;[21]磁盘类型标识符(默认 0xf0， 3.5 寸软盘。0f8h 表示硬
盘。）
    DW  9               ;[22]FAT 长度（9，OS 扇区=表数*扇区数+隐藏扇区数）
    DW  18              ;[24]每个磁道扇区数(默认为 18，软盘默认值为 18，此处须
为 18）
    DW  2               ;[26]磁头数(磁盘面数，软盘为 2 面，正反面各一个，此处
须为 2）
    DD  0               ;[28]隐藏扇区数(默认为 0。此字段参与根目录、用户数据
位置计算）
    DD  0               ;[32]分区/磁盘逻辑扇区总数（大于 65536 用此字段，否则
19）
    DB  0,0,0x29        ;[36]物理驱动器号、[37]未使用位、[38]扩展引导标志
    DD  0xffffffff      ;[39]卷标号码（卷序列号、磁盘号，默认为 0）
    DB  "NNOS_OS    ";[43]磁盘名称(卷标、卷名称，必须为 11 字节，不足则以空
格填充）
    DB  "FAT12   "      ;[54]文件系统类型(必须为 8 字节，不足则以空格填充）
    ;#RESB  18          ;[62]待用,引导代码，由偏移 0 字节处的跳转
来）
    TIMES   18  DB 0    ;[62]#[兼容 nasm][代]
```

（3）程序核心入口。程序核心入口标签为 entrry。用于寄存器的初始化。

（4）读磁盘。IPL 完成定义、声明和寄存器初始化后，开始读盘，从 C0-H0-S2（S1 的 512 字节为 IPL 本身，已被 BIOS 装入内存）开始读，将读取磁盘数据装入内存 0x0820 开始的区域。本部分有三个标签，分别为 readloop、retry 和 next。读盘程序被分割为 4 部分，读盘成功时，不断执行 next 读取下一个扇区，失败时则跳转到 readloop，同时错误计数器加 1。读到最大磁盘柱面号后，跳转到 showmsg 显示系统信息。showmsg 将 msg 程序段首址写入 SI，msg 程序段用 DB 命令写入字符。字符属性以 8bit 指定，从高位到低位依次为：闪烁与否(1 位)、背景色(3 位。)、高亮与否(1 位)、前景色(3 位)。然后跳转到 putlooop 将字符逐个写入显存。

写入完毕跳转到 start，start 将磁盘内容的结束地址写入内存供 OS 使用，跳转到 OS 起始位置 0x7c00（0x00007c000~0x00007dff 为 IPL，与软盘拷

贝位置一致)。若引导失败，则跳转到 error，打印出错误信息。

3.4.2 SYSHEAD 详细设计

SYSYHEAD 为系统实模式部分，完成操作系统运行的准备工作。

（1）处理器版本声明（仅 nask)。声明采用的的指令集(80486)，以支持 LGDT、EAX、CR0 关键字,仅适用于 nask， nasm 中无需声明。声明在 nasm 中为注释状态，编译前，Makefile 调用 getnas.exe 将 asm 文中被注释的 nask 代码去掉注释、替换不同语法关键字，转为 nask 可用的.nas 源文件（其他.asm 转.nas 源文件的过程与此同)。

（2）BOOT 信息存储位置声明。实模式下声明了常量和 BOOT 信息在内存的保存位置，并将相应信息写入供保护模式使用。包括 BOOT 加载位置、缓存首址、cache 首址、最大磁盘柱面号位置、LED 灯状态位置、显卡模式位置、屏幕横（纵）分辨率保存位置、屏幕纵向分辨率保存位置和图像缓冲区保存位置。如表 5、表 6 所示。

表 5 常量声明

键	值	含义
BOTPAK	0x00280000	bootpack 加载位置保存位置
DSKCAC	0x00100000	磁盘缓冲区首地址保存位置
DSKCAC0	0x00008000	cache 首地址保存位置
VBEMODE	0x105	VBE 显卡模式切换指令 (1024*768)

表 6 BOOT 信息保存位置

键	值	含义
CYLS_MAX	0x0ff0	最大磁盘柱面号保存位置
LEDS	0x0ff1	键盘 LED 指示灯状态信息保存位置
VMODE	0x0ff2	显卡模式位数信息保存位置
SCRX	0x0ff4	屏幕横向分辨率信息保存位置
SCRY	0x0ff6	屏幕纵向分辨率信息保存位置
VRAM	0x0ff8	图像缓冲区首地址信息保存位置

[注]本表表项与 BOOT 信息结构体一一对应。

（3）系统装载位置声明。以上声明之后，又指明了操作系统的装载位置 0xc200。软盘文件名存于 0x2600 后，内容在 0x4200 后，启动区位于 0x8000，则系统装载到 0x8000+0x4200=0xc4200。

（4）确认机器是否支持 VBE 标准。根据 VBE 标准，显卡 VBE 信息须写入内存 ES:DI 开始的 512 字节区域。本程序先将 0x9000 写入 ES（需要通过 AX 中转），然后将 0 写入 DI。将 AX 赋值为 0x4f00，若不支持 VBE，赋值失败。用 JNE 指判断赋值结果，成功则继续执行，否则说明机器不支持 VBE，跳到 scr320，用 VGA 的 320×200×8 低分辨率模式运行操 OS。

（5）检查 VBE 版。VBE 版本须为 2.0 以上，非则跳到 scr320，是则继续执行。

（6）获取 VBE 的画面模式相关信息。先将显卡模式信息地址写入 CX，再将模式 0x40f01 写入 AX，调用 BIOS 中断，比较 AX 与 0x40f01，等则继续，否则跳到 scr320。模式信息位置见附录Ⅰ表 14。

（7）画面信息确认。依次判断颜色位数是否为 8、指定方式是否为调色板模式、模式属性第 7bit 是否为 1，若以上全部确认无误，则继续向下执行。有任何一项不符，则跳转到 scr320。

（8）画面模式切换。执行至此，表明机器支持 VBE，将指令偏移 0x4000（规定）写入 BX，再将选定模式写入 AX 并调用 BIOS 中断，将 VBE 模式信息从附录Ⅰ表 14 的对应地址取出，存在正文表 7 定义的相应内存位置，供保护模式下初始化 BOOT 信息时使用。

3.4.3 OSFUN 详细设计

本模块完成敏感操作，包括 I/O 端口读写、EFLAGS 读写等。对应的源文件为 osfun.asm/.nas。

本模块函数名前面均须加下划线，并用 GLOBAL 声明（调用 C 语言中断处理函数时，用 EXTERN 声明引入），以供编译器链接，完成汇编与 C 语言的混编。本模块函数列表及简介见附录Ⅵ表 40，具体实现分析略，详见 osfun.asm。

至此，整个引导层详细设计介绍完毕。

3.5 基础层详细设计

基础层包括 GDT&IDT、INTERRUPT、Manager Interface、TIMER 和 FIFO 五个模块，对应 gdiidt.c、interrupt.c、nnos.h、timer.c 和 fifo.c 五个源文件。实现 GDT、IDT、中断处理、定时器、FIFO，并通过 Manager Interface 为 OS 和应用程序提供服务。

3.5.1 GDT&IDT 详细设计

本模块完成 GDT 和 IDT 的初始化和中断处理函数注册。函数表见附录Ⅵ表 41。setSegmDesc()参数依次为 GDT 变量、段上限、段基址和段访问权限。setGateDesc()的参数依次为 IDT 变量、段偏移、段选择子和段访问权限。

（1）initGdtIdt()函数。本函数先定义 GDT 和 IDT 首地址，再遍历 GDT 所有段并初始化，段属性置 0，设定全局段（段上限为整个寻址空间）基址和权限，上限置为 FULL_ADR，基址置为 ZERO_ADR，权限置 DATE32_RW_PRE（应用，

可读)。设定 BOOT 段，段上限为 BOOT_LIMIT，基址为 BOOT_ADR，权限为 CODE32_ER_PRE（系统，可执行)。然后调用 osfun.asm 的 load_gdtr()，将 GDT 上限、基址传入，GDT 信息写入 GDTR。

GDT 初始化后，初始化 IDT。遍历 IDT 段，段属性置 0。调用 osfun.asm 中的 load_idtr()，将 IDT 上限、基址传入，IDT 信息写入 IDTR。完毕后注册中断处理器，调用 setGateDesc()，将指定 IDT 段、中断处理器、段选择子和权限在 IDT 注册。

本系统目前注册了 asm_inthandler20、asm_inthandler21、asm_inthandler27、asm_inthandler2c 和 asm_sys_api，其函数及功能简介见表录Ⅵ表 1。

（2）setSegmDesc()函数。本函数先判定对段合法性，若段上限大于 0xfffff，为应用，调整权限和上限。此后将段上限、段基址、段访问权限赋给 GDT 变量的相应成员，完成段设定。相关规定见概要设计。具体实现及注释见 gititd.c。

（3）setGateDesc()函数。本函数对段属性信息处理后，赋给 IDT 结构体变量中的成员。详见 gditidt.c。

3.5.2 INTERRUPT 详细设计

限于页数，略（下同)。函数设计见附录Ⅵ表 42，实现见 interruput.c 及其注释。

3.5.3 TIMER 详细设计

函数设计见附录Ⅵ表 48，函数实现见 timer.c 及其注释。

3.5.4 FIFO 详细设计

函数设计见附录Ⅵ表 43，函数实现见 fifo.c 及其注释。

3.5.5 Manager Interfac 详细设计

对应源文件 nnos.h 在根目录下 lib 子目录，含全部函数声明和 lib 下的 sysdef.h、systructual.h，其他源文件使用系统常量或结构体，只需包含 nnos.h。有关常量、数组等定义见 6.2 节，结构体定义 6.3 节。所有函数声明简介见附录Ⅵ表 40~55。

本模块使用汇编语言编写了 API 供应用程序使用。API 设计略。通过 32 位寄存器向系统 API 中断处理函数 sys_api()传值，sys_api 根据所传值调用相应的系统函数为应用程序服务，包括申请内存、释放内存、创建窗口、打印字符串、绘制图形等。

系统 API 提供了一般异常保护和栈异常保护功能。任何应用程序试图非法

访问系统段，系统将产生一般异常保护中断，阻止应用程序越权访问，并在控制台输出处理信息。任何应用程序与栈关的错误（例如数组越界），都将触发系统栈异常保护中断，使之无法对操作系统造成损害。

系统 API 还提供了应用程序强制终止功能。因本系统采用时间片轮转实现多任务，所以有问题的应用程序（比如无意义的无限循环）会浪费系统资源，造成系统卡顿，影响其他应用程序运行。此时，可以调用强制终止功能，结束应用程序。对应的功能键为 Shift+F4。

3.6 核心层详细设计

核心层含四大模块、UI 和 BOOT，完成 OS 核心功能。分别对应 memery.c、devices.tss.c、filemanage.c、graphics.c 和 bootpack.c、task.c、file.c。函数表见附录Ⅵ。

限于页数，本节仅对部分模块的部分函数作简述。所涉常量、函数见附录。

3.6.1 Memery Manager 详细设计

本模块完成内存管理，简介见 5.3.4 节。系统维护一张内存空闲块表，该表首地址定义于系统常量中。系统和应用申请内存或释放时，内存管理系统根据此表完成操作并同步变动。原则上，在整个系统运行周期内，此表有且仅有一张。

（1）内存检查函数（memeryCheck()）。参数：start（内存起始地址，整型），end（内存结束地址，整型）。调用 OSFUN 的相关服务读取 EFLAGS 的信息并校验，然后调用 OSFUN 的内存检查子函数（memetest_sub()）完成内存检查，返回检查结果（内存总容量），非 0 为正常，0 为异常。

（2）初始化内存表函数（initMemeryManager()）。参数：memeryList（内存表，MEMERY_LIST 结构体类型，下同）。本函数对 memeryList 的各个成员进行初始化。调用 memeryCheck()获得内存总量赋值给 total_size。其他成员皆初始为 0。

（3）计算剩余空间函数（memeryTotal）略。

（4）内存分配函数（memeryList）。参数：memeryList，size（申请内存块大小，无符号整型）。首先调用 sortMemery()重新排序，确保空闲块按容量递增排序。然后遍历 memeryList，寻找第一个容量大于或等于 size 的空闲块，若恰好相等，则直接调用 deleteBlock()在 memeryList 中删除此块并将此块首地址返回。若大于 size 则偏移块首地址、更新块容量，然后将原来的块首地址返回。最后重排内存块。

（5）内存释放函数（memeryFree()）。参数：memeryList，addr（内存块首地址，无符号整型），size。本函数首先调用 sortAddr()，将内存块按地址递增排列，遍历 memeryList 寻找第一个地址不小于待释放块的块。分析该块的前后块，若前有可用块，判断该块前一块尾地址加待释放块容量是否恰为该块首地址，是则将三块合并，调用 DeleteBlock()删除该块，并调整合并后新块的属性。若前无可用块，则判断待释放块首地址加容量（尾地址）后是否恰为

该块首地址，是则两块合并，调整新块属性；否则调用 addBlock()增加新空闲块，将新块容量与历史最大块地址比较，更新数据。若该块恰为第 0 块，则判断带释放块尾地址是否恰为该块首地址，是则合并两块，更新块属性；否则增加新块，更新数据。若该块恰为最后一块，则判断该块尾地址是否恰为待释放块地址，是则合并两块；否则增加新块，将带释放块放在最后。最后更新 memryList 中记录的内存总量，调用 sortMemery()，将内存块按容量递增重排。

（6）4K 取整方式申请内存函数（memeryAlloc4k()）。参数：与 memeryAlock 同。本函数对申请的空闲块进行调整，使之大小最小为 4KB 或恰好能被 4KB 整除，从而减少碎片并使逻辑空间与物理空间契合，称 4K 取整。公式为：

$$size = (size + 0xfff) \ \& \ 0xfffff000 \tag{3}$$

（7）4K 取整方式释放内存函数（memeryFree4k()）。原理同上。

（8）删除内存表项(deleBlock())和增加内存表项(addBlock())函数说明略。

（9）内存块递增排序（sortMemery()）和地址递增排序（sortAddr()）函数说明略。两函数皆才使用简单但低效的冒泡排序算法[16]。

3.6.2 I/O Manager 详细设计

系统通过中断响应机制响应 I/O 中断，完成内存管理。简介见 5.3.4。本模块对 I/O 中断进行处理，对中断数据进行解析，并调用同层的 Graphics 中的相关函数完成信息图形化展示。对应源文件为 devices.c 和 interrupt.c 的 I/O 中断处理器部分。本模块以来 FIFO 模块建立先进先出缓冲区。本节仅详细说明键盘鼠标处理，其他略去。

（1）鼠标中断解码函数（decodeMouse()）。参数：mouseCursor（鼠标指针缓冲区首地址，MOUSE_CURSOR 型指针），data（鼠标码，无符号字符型）。鼠标中断数据第一个字节为标志位，无意义，故在完成判断后应舍去。三个字节构成一个鼠标码，故函数维护一个模为 3 的 phase，phase 为 0 时，判断 data 是否为标志位，是则将 phase 置 1。对每个有效字节，接收后须与 0x08 作与运算以校验数据是否正确，不正确则抛弃，phase 不变；正确则 phase 加 1，继续等待后面的字节。根据规定，鼠标码第 1 位横向移动量（mx），第 2 位为纵向移动量（my），低三位为按键状态（mbutton）。满三个字节后，解析鼠标数据。解析公式分别为：

$$mx \ |= 0xffffff00 \tag{4}$$

$$my \ |= 0xffffff00 \tag{5}$$

$$mbutton = dataBuffer \ \& \ 0x07 \tag{6}$$

最后返回解析状态，成功为 1，失败为 0。

（2）本模块其他函数说明略，见 devices.c。

3.6.3 Process Manager 详细设计

系统维护多个优先级队列，不同队列有着不同优先级。各队列都有多个任务，各任务也有不同优先级。优先级高者先执行，且分配更多时间片。对应源文件为 task.c。

（1）任务初始化函数（initTaskList()）。参数：memeryList。首先定义 gdt，首地址为 GDT_ADR，调用 memeryAlloc4k()为 taskList（任务列表，下同）申请内存空间。然后遍历任务列表，将所有任务项的状态标志置 0（未分配），并从 TASK_GDT0 开始，以 8 位为单位，为所有表项分配 gdt 段、设定段属性。设定完成后，遍历 taskList 中的 task_level（优先级队列），将所有优先级队列的任务计数器、当前任务号置 0。以上工作完成后，调用 taskAlloc()为默认任务（原则上为 BOOT）申请任务项，并置任务状态标志为 TASK_FLAG_RUNNING。设其优先级为 2（时间片 0.02 秒），挂在在第 0 队列（最高优先级队列），调用 taskSwitchSub()切换任务。然后调用 OSFUN 的 load_tr()将任务段选择子装入 tr 寄存器，再调用 timerAlloc()为 taskTimer（任务计时器，用于时间片，下同）申请空间，并设其超时时间为任务优先级。最后，创建一个 idle（空闲任务），为其申请内存空间，设置为最低优先级。用于在系统中无任务运行后，唤醒处于休眠状态的任务，以防死锁。尽管采用 FIFO，但本系统暂不考虑 Belady 异常[17]。

（2）任务自动切换函数(taskSwich())。无参。函数维护一个 taskLevel（TASK_LEVEL 结构体类型指针），该指针始终指向 task_level 中当前正在运行的任务。每调用一次本函数，当前正在运行的任务号偏移 1，模为当前优先级队列任务数。判断此时任务列表的任务变化状态位，非 0 则调用 taskSwichSub()切换，优先级队列，并将 newTask 指针指向优先级队列任务列表中正在运行的任务，将 taskTimer 超时时间为 newTask 的优先级，判断当前是否有两个以上任务，是则跳转到 newTask 执行。

（3）任务休眠（taskSleep()）。参数：task。首先判断 task 状态位是否为运行状态（TASK_FLAG_RUNNING），是则将 myTask 指针指向当前任务，调用 removeTask()将 task 从任务列表移除。再判断 task 是否就是 myTask，是则为自我休眠，需要切换一个新任务继续运行。

（4）本模块其他函数略，见 task.c。

3.6.4 FileManager 详细设计

SYSHEAD 切 BOOT 前，会读磁盘到 CYLS_MAX，此过程中，FAT 也被装入了内存。系统基于 FAT 完成文件管理。

（1）FAT 解码引擎（readFAT()）。参数：fat（FAT 缓冲区首地址，整型指针），img（磁盘首地址，无符号字符型）。大于 512 字节的文件，存放于不连续的扇区中。最小连续扇区单位称簇，本系统每簇一个扇区。文件所有簇号存放于 FAT 中，首地址为 C0-H0-S2，共 9 个扇区。管理的磁盘范围为 0x0002~0x0013ff。本函数从 FAT 首地址开始遍历 FAT，使用微软提供的 FAT 解码算法每三个字节一组对 FAT 解码，解码结果存入 fat。

（2）文件装载器（loadFile()）。参数 cluster（簇号，整型），size（簇

大小，整型），file（文件缓冲区首地址，字符型指针），fat，img。本函数首先判断扇区大小是否小于 512，是则直接装在该簇数据，否则遍历所有簇并装载。每循环一次，size 减少 512，文件缓冲区向后偏移 512。

（3）查找文件函数（serchFIle()）。参数：fileName（文件名首地址，字符型指针）。函数维护文件信息指针 fileInfo，指向磁盘首地址 DISK_ADR 偏移 0x002600（文件数据区位置）处。然后初始化数据游标 i 和 j，将标志位 flag 置 0，表示尚未找到。然后调用 convertToUppercase()将文件名转为大写。遍历文件信息区域，查找文件，若当前 fileInfo 的文件名与目标文件名相同，则 flag 置 1，返回 fileInfo 的地址。

3.6.5 Graphics 详细设计

本模块负责图形绘制、图形显示，图形处理已分离为独立的图层模块。具体实现的源文件为 graphis.c。依赖 sysdef.h 中的颜色表和多个结构体。相关函数及功能简介参看附录Ⅵ表 45。

（1）画板设定函数（setPalette()）。参数：start（起始色号，整型）、（终止色号，整型）和 rgb（无符号字符型指针）。首先保护现场并屏蔽中断，以防设定失败。然后调用 OSFUN 的 io_out8()服务，向端口 0x03c8 写入 start，遍历颜色表，将对应 RGB 十六进制数右移处理后，三位一组写入端口 0x03c9。最后恢复现场。

（2）画板初始化函数（initPalette()）。本函数调用 setPalette()，完成换班初始化。

（3）桌面初始化函数（initDesk()）。参数：显存或缓冲区首地址（vram，字符型指针，下同）、屏幕或缓冲区宽度（scrx，短整型，下同）、屏幕或缓冲区高度（scry，短整型）。定义了桌面色块数组（deskWindows，使用 WINDOW 结构体，在定义时初始化），并将该数组和 vram、scrx、deskWindows 一并传入 boxDrawx()，绘制出桌面。

（4）像素点渲染（pointDraw8()）。参数：vram、scrx，像素点（point，PIX_POINT 结构体类型）。根据参数，直接对数组 vram 赋值指定位置颜色值，该值记录在 point 的 vcolor 中。

（5）色块渲染（boxDraw8()）、图形渲染（pictureDraw8），略。

（6）色块批量渲染引擎（boxDrawx）。参数：vram、scrx、色块数组指针(box，BOX 结构体类型，下同)。本函数有三层循环，第一层遍历 box，第二层和第三层从指定的显存或换成首地址开始先上后下、先左后右地扫描该区域并写入颜色。

本函数具体实现代码块如下：

```
/*色块批量渲染*/
void boxDrawx(char *vram,short scrx,BOX *box){ //矩形区域绘制函数
    int vx,vy; //用于遍历的像素坐标
    while(box->bx0 != -1){
        for(vy = box->by0;vy <= box->by1;vy++){
            for(vx = box->bx0;vx <= box->bx1;vx++){
```

```
                vram[vy * scrx + vx] = box->vcolor; //根据行列计算出地址
偏移量，将 vclolor 写入以 varm 为基地址、以 y*width+x 为偏移量的地址中，即显存的
相应位置
            }
        }
        box++; //下一个色块
    }
    return;
}//本函数三个参数分别为显存或缓冲区地址、屏幕或缓冲区宽度和色块数组首地址
```

（7）字体渲染引擎（fontDraw8()）。参数：vram、scrx、fx（字体宽度，整型）、fy（字体高度，整型）、vcolor（色号，字符型）、font（字体首地址）。点阵字体为 16×8 位，本函数设有一个计数器 i，用于自上而下以 8 位为单位扫描字体。通过移动指针 p 遍历显存区域，每次对字体的 8 个二进制为进行校验，非 0 便将颜色信息写入该显存区域。显存区域首地址计算公式为：

$$p = vram + (fy + i) * scrx + fx$$

（7）

（8）窗口绘制函数（windowDraw8()）。参数：vram、scrx、window（WINDOW 结构体类型，下同）、focus（焦点标志，整型）、area（刷新区域标志，整型）。本函数首先判断窗体类型（window.type），根据所指定的窗体类型，将相应系统默认窗体定义中的相应色块数组首地址赋值给 box 指针。然后将 window 中窗体信息（长、宽等）赋值给 box 数组中的窗体色块，并为窗体添加四周边线、关闭按钮。然后根据焦点状态和刷新区域决定窗体标题栏的着色，根据 window.caption 调用 wordsDraw8()绘制窗口标题。

（9）创建自动窗口（createWindow）、刷新窗口标题（refreshWindowCaption）、标签绘制（labelDraw）、文本框绘制（makeTextBox8()）函数略。

（10）鼠标图像点阵解析引擎（initMouseCusrsor8()）。参数：cursorGraph（鼠标点阵数组首地址，字符型指针，下同）、curBackColor（鼠标指针背景色，字符型）。本函数从左到右、自上而下扫描 cursorGraph，判断相应数组元素的字符类型，将“*”解析为指针黑色边线，将“1”解析为指针白色填充，将“0”解析为 curBackColor。

3.6.6 BOOT 详细设计

本模块为系统主程序，对应文件为 bootpack.c，是系统保护模式运行的起始位置。系统主函数入口规定为 NNOSMain()，系统和应用程序源文件都必须实现此函数，否则无法运行。本函数定义了 BOOT 信息首地址、通用 FIFO 缓冲区、缓冲区数组、提示信息缓冲区、键盘控制电路缓冲区、屏幕信息、鼠标指针指针信息、鼠标指针图像缓冲区、内存表、图层列表、控制台、任务列表和定时器列表等。完成上述内容的初始化和设定工作，然后进入永真循环，正是开始为用户提供服务。调用基础层服务，接收并解析 I/O 缓冲区数据，完成信息展示和命令执行。

3.7 扩展层详细设计

拓展层往上为外壳。本层共六个模块，概要设计已说明。本项目只实现FONT BASE、Picture Drawer 和 Layer Drawer。详略，见附录Ⅵ。

3.8 应用层详细设计

本层面向用户，提供基本应用程序，包括 Console、Task Explore、File Explore、GB2312 Support 和其他应用五个模块。详略，见附录Ⅵ。系统 U 盘启动在本节说明。

3.8.1 Console 详细设计

Console 完成了文件查看和可执行文件运行，命令格式兼容 Windows 风格和 Linux 风格。实现了 dir（ls）、type（cat）、mem（free）、run、version、sysinfo、cls(clear)命令。支持.nex 格式可执行文件运行。函数设计见附录Ⅵ表 51。本节仅介绍部分函数。

（1）控制台换行函数（newCMDLine()）。参数：console（CONSOLE 结构体类型指针，下同）。首先判断当前控制台光标纵坐标是否已经到达允许输入的倒数第一行，否则将光标纵坐标下移一行。若光标纵坐标已经到达倒数第一行，则遍历整个控制台图层的文本输入区域，自上而下、从左到右将所有像素上移一行。然后将最后一行刷新为空行，实现控制台自动滚动。最后将光标移动到开头的控制台提示符右侧。

（2）控制台输出函数（sysprint()）。参数：console，charCode（整型），movFlag（字符型，光标移动标志位）。函数维护一个两字节的字符缓冲区 info。将 charCode 赋值给 info[0]，将“\0”赋值给 info[1]。然后判断 info[0]是否为特殊字符（制表符、换行符、回车符）等。对于回车符不作处理（兼容 Linux 文件），对于换行符，调用 newCMDLine()换行。对于制表符，则以 4 位基本单位寻找制表位，使用 lableDraw 填充空格。若为普通字符，则使用 labelDraw()在控制台图层上绘制出字符。

（3）运行可执行文件函数（rumCMD()）。参数：console，fat，command（命令首地址，字符型数组指针）首先定义 memeryList 并设首地址为 MEMERY_ADDR，指向系统内存表。然后定义 GDT 并设首地址为 GDT_ADR，指向系统 GDT。函数维护一个文件指针 file,，一个数据指针 data，一个任务指针 task。首先初始化游标 x，y，将信号 flag 置 0，表示文件名非法或不合规范（没有拓展名，截取文件名前八个字节）。然后向文件名中填充空格使之满 8 字节。若无拓展名，则补充拓展名“NEX”。之后，将文件名和拓展名连接，形成 11 字节的文件全名。调用 searchFile()，在磁盘查找此文件。找到则将此文件信息首地址赋值给 fileInfo。fileInfo 非 0，则为 file 申请内存空间，并将用用程序内容段首地址设为 0xfe8.然后调用 laodFIle()加载文件内容。加载完成后，判断 fileInfo 的大小是否符合要求、可执行文件标志“NNOS”是否存在、文件第一个字节是否为 0x00，验证通过后，设置与文件相应的 gdtSize、

esp、dataSize 并为 data 申请内存空间，设置其段地址。然后设定应用程序段，调用 OSFUN 中的 run_app()运行程序。运行完毕后调用 memeryFree4k()释放应用程序所占空间。

3.8.2 其他应用详细设计

从略。

3.8.3 系统 U 盘启动详细设计

本系统 U 盘启动后在物理机运行的情形如图 36 所示。

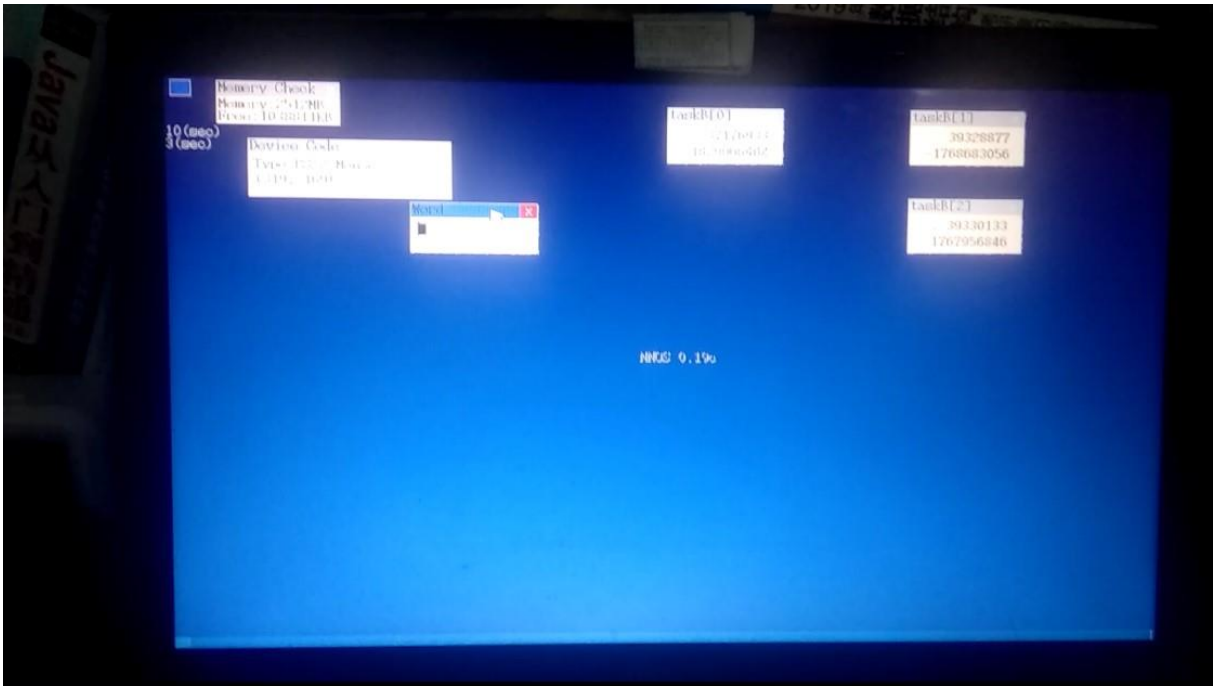


图 36 正在物理机中运行的 NNOS_0.19c

本系统支持软盘启动和 U 盘启动。U 盘启动实现步骤为：①编译生成 nnos.img；②格式化 U 盘为 FAT32；③将 grub 安装到 U 盘；④设置 grub 配置文件；⑤将 nnos.img 复制到 U 盘；⑥使用 Defraggler 整理 U 盘碎片；⑦关机，插入 U 盘；⑧开机，进 BIOS 设置 U 盘为首先启动，保存；⑨启动成功。

grub 配置文件 memu.lst 配置如图 37 所示。

```
1  map --mem (hd0,0)/nnos.img (fd0)
2  map --hook
3  chainloader (fd0)+1
4  rootnoverify (fd0)
```

图 37 memu.lst 配置

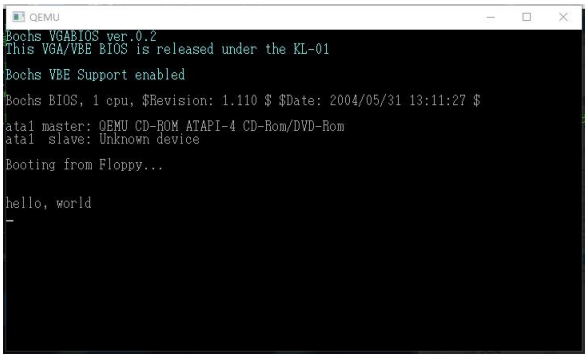
四 测试分析

测试的目的在于排除潜在系统漏洞、检验新版本系统性能。测试中反应出的问题可作为系统迭代的修正目标，性能信息可为内核优化提供参考依据。本系统开发过程中，每个版本的新增功能都要进行测试，这些测试没有必要全部在本节说明，本节仅对重要测试及其结果进行分析。

4.1 版本迭代测试

（1）0.00 版本测试。

本版本有纪念意义，开发 OS 的第一步，异常艰难。本版本直接用机器语言编写，功能为开机后在屏幕显示“hello，world”。在 QEMU 中运行测试正常，如图 38 所示。



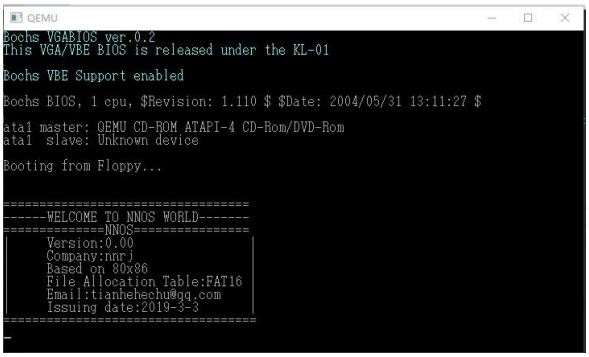


图 38 正在 QEMU 虚拟机中运行的 NNOS_0.00（左）、图 39 正在 QEMU 虚拟机中运行的 NNOS_0.02a（右）

（2）0.02 版本测试 。

本版本开始用汇编开发并拓展输出信息，后由 IPL 继承。测试结果正常，如图 39 所示。

（3）0.06j 版本测试。IPL 开发完毕，SYSHEAD 完成字符显示模式切换到绘图模式。黑屏是因程序尚未向显存中写入任何信息。测试正常。如图 40 所示。

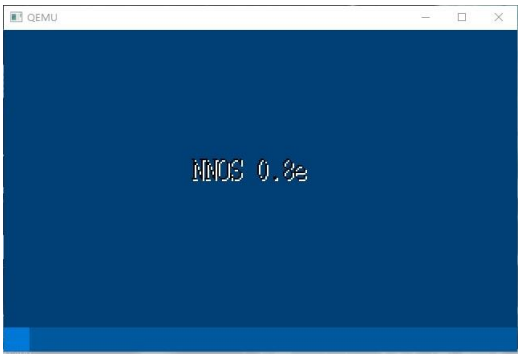
（4）0.07e 版本测试。本版本在屏幕上绘制彩色条纹。测试结果正常。如图 41。

（5）0.08e 版本测试。绘制出桌面和版本号信息。测试正常如图 42 所

示。

（6）0.09e 版本测试。绘制鼠标、响应键盘中断、键盘码输出。每按一个键，左上角显示的键盘码就会发生变化。测试正常，如图 43 所示。

图 40 正在 QEMU 虚拟机中运行的 NNOS_0.06i（左）、图 41 正在 QEMU 虚拟机中运行的 NNOS_0.07e（右）



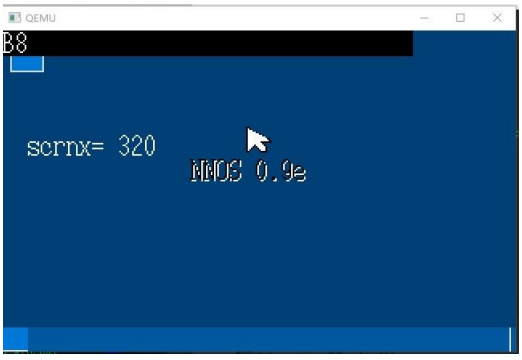
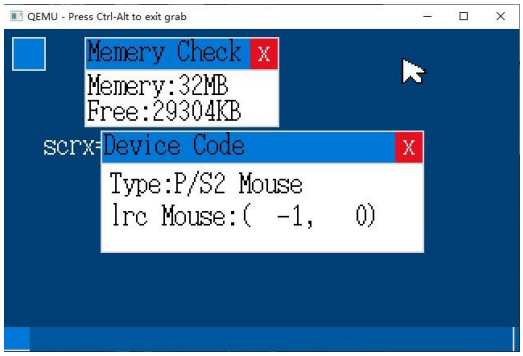


图 42 正在 QEMU 虚拟机中运行的 NNOS_0.08e（左）、图 43 正在 QEMU 虚拟机中运行的 NNOS_0.09e（右）

（7）0.12d 版本测试。实现了窗口绘制、内存检查、内存管理和鼠标指针

移动。在屏幕上绘制内存检查窗口，显示出当前内存，随着鼠标移动刷新桌面。测试结果异常，鼠标移动正确，但经 LRC 校验后，鼠标坐标显示错误，此问题在后续版本中得到了修复。如图 44 所示。



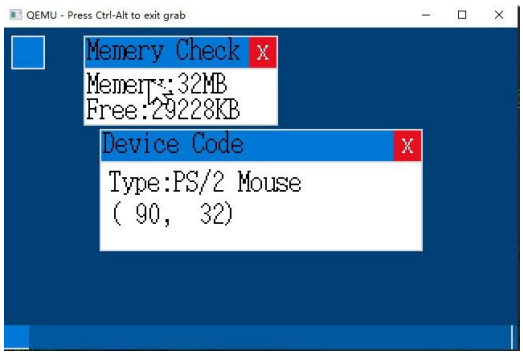
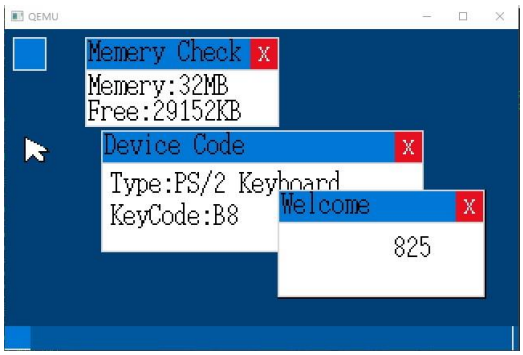


图 44 正在 QEMU 虚拟机中运行的 NNOS_0.12d（左）、图 45 正在 QEMU 虚拟机中运行的 NNOS_0.13b（右）

（8）0.13b 版本测试。实现了图层处理（在此之前，鼠标移动会将已绘制的窗口擦除），修复了鼠标坐标显示错误。测试结果正常。如图 45 所示。

（9）0.14h 版本测试。本版本实现了定时器。屏幕上创建一个新窗口，显示定时器计时信息。计时器每秒增长 100。测试结果正常。如图 46 所示。

（10）0.15g 版本测试。本版本对 3 秒定时器、10 秒定时器和光标闪烁(间隔为 0.5 秒)进行测试，测试结果正常。如图 47 所示。



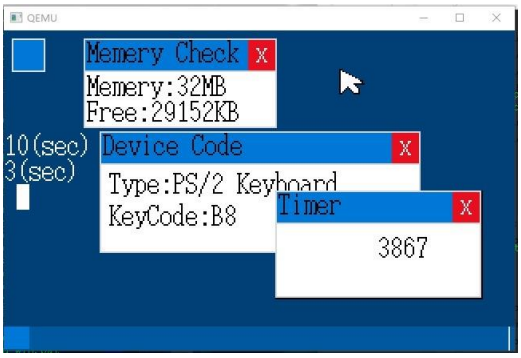


图 46 正在 QEMU 虚拟机中运行的 NNOS_0.16h（左）、图 47 正在 QEMU 虚拟机中运行的 NNOS_0.15g（右）

（11）0.17i 版本测试。本版本实现了 VBE 高分辨率支持、窗口移动和文本输入。分辨率为 1024×768，色彩模式为 8 位。将 Word 窗口从初始位置进行了拖动，在文本输入框中键入了“TEST”，测试结果正常。如图 48 所示。

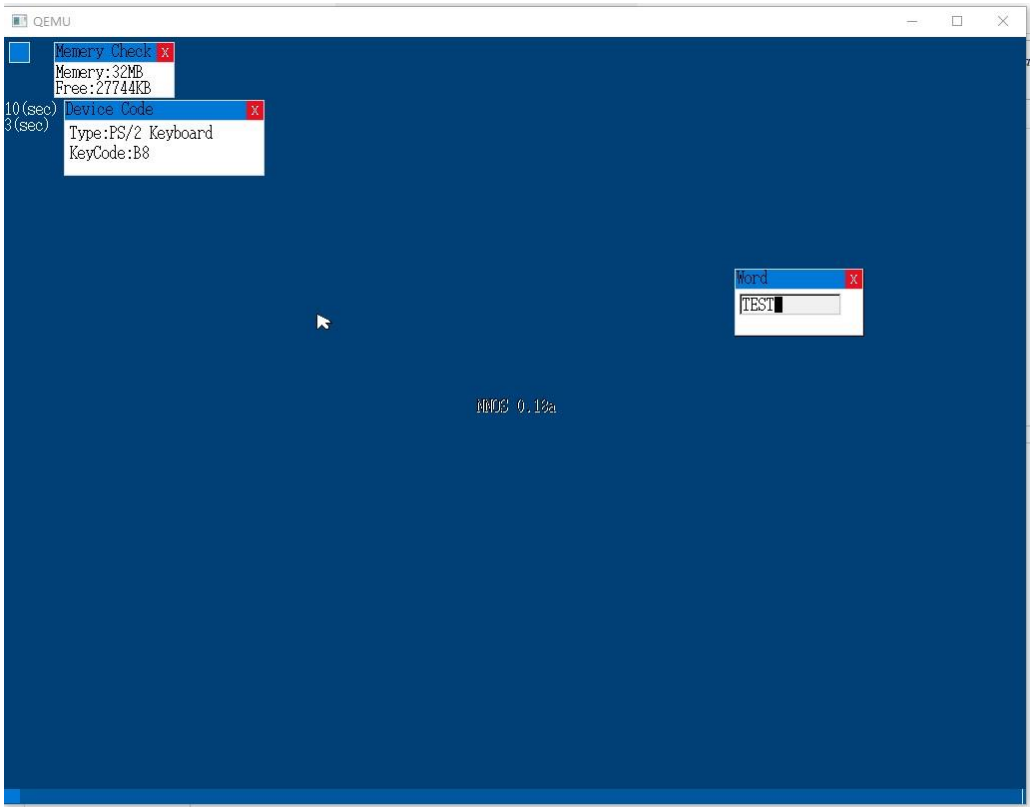


图 48 正在 QEMU 虚拟机中运行的 NNOS_0.17i

（12）0.24d 版本测试。本版本基本实现 OS 四大核心模块。用控制台执行 ls、运行了件 hello.nex（它调用 API 输出信息）。内容超出自动滚屏。测试正常，如图 49。

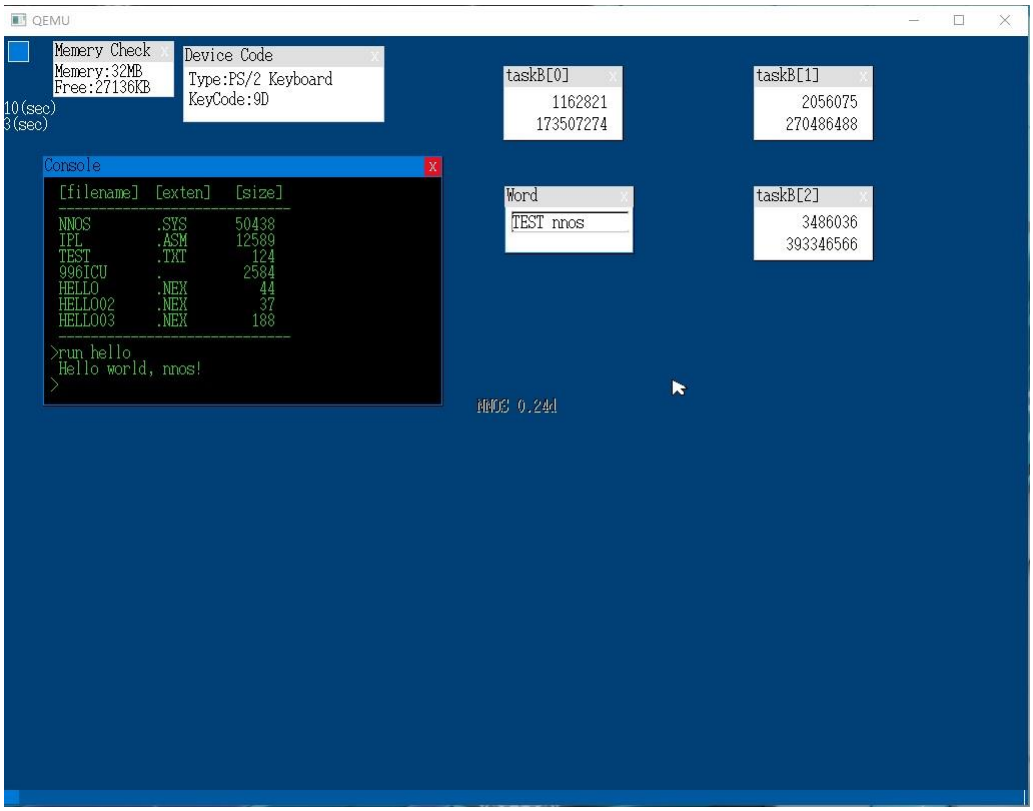


图 49 正在 QEMU 虚拟机中运行的 NNOS_0.24d

至此，历史迭代版本测试结果分析完毕。

4.2 定时器性能测试

0.16a 到 0.16h，对定时器进行包括数据结构、算法在内的内核级别优化。本节对优化前后的性能测试结果进行分析对比，优化前后的版本各进行五次测试。

（1）优化前版本五次测试结果图 50 所示。

图 50 优化前定时器测试结果

测试项目：定时器测试；系统版本 0.16a；测试用机器：qemu 虚拟机；

测试方法：在中断处理前执行累加，10 秒后输出累加值(3 秒归零)，去掉停机等待指令，CPU 全速运行；测试结果：如表 7 所示。

表 7 优化前定时器测试结果

序号	值
1	31494132
2	36872068
3	37824938
4	37075437
5	36979883

数据分析：均值 36049291.6；

测试结论：累加值在 37000000 上下波动，波动较大，稳定性不合预期。虚拟机受所在物理机影响较大，将来可在物理机中重新测试。

（2）优化前版本五次测试结果图 51 所示。

图 51 优化后定时器测试结果

测试项目：定时器测试；系统版本 0.16h；测试用机器：qemu 虚拟机；

测试方法：同上。 测试结果：如表 8 所示。

表 8 优化后定时器测试结果

序号	值
1	70340699
2	70346727
3	71440518
4	68810673
5	66747264

数据分析：均值 69537176.2，相对 0.16c 性能提升 1 倍（为原来的 1.9 倍)；测试结论：①累加值在 70000000 上下波动，波动较大，稳定性不合预期。虚拟机受所在物理机影响较大，将来可在物理机中重新测试。②累加值在同样的测试时间内，相对于优化前版本大约翻了一倍，优化后定时器性能明显提高。

4.3 内核漏洞列表及详情

本系统开发过程中，出现过一系列 BUG，不乏严重的内核漏洞。典型的内核漏洞，全部记录在系统漏洞列表，永久跟随系统迭代。这些漏洞无不曾使作者焦头烂额，有纪念意义。更重要的是对后期完善系统、协同开发有巨大参考价值。

4.3.1 显存污染漏洞

编号：KB00000000。状态：已修复。命名：薛定谔。出现版本：不可考。
描述：显存初始化后，部分寄存器未初始化，导致显示器显示不可预知的色块。
解决方案：初级解决方案，绘图前用背景色覆盖整个显存;。终极解决方案，重写绘图函数，修复地址错误。
发现者：年年软件；发现日期：2019.3.6；修复日期：2019.3.7。

4.3.2 中断处理器无响应漏洞

编号：KB00000001。状态：已修复。命名：梦魇。出现版本：不可考。
描述：中断处理器无响应，键盘不可用。
原因分析：中断描述符错误。
解决方案：终极解决方案，修正内核中断描述符错误。
发现者：年年软件；发现日期：2019.3.8；修复日期：2019.3.9。

4.3.3 中断处理崩溃无限重启漏洞

编号：KB00000002。状态：已修复。命名：黄先生。出现版本：NNOS 0.10f。
描述：开机无限重启。此漏洞影响大部分机器，少部分机器正常运行。
原因分析：gdtigt.c 中 initGdtIdt()函数初始化 GDT、IDT 失败。初始化过程中，GDI 和 IDT 的参数范围传输错误，造成 0.10f 版本后 IDT 地址空间不足，增加鼠标中断后，无法注册中断处理器 inthandler27，导致针对 PIC0_IRQ7 的中断处理补丁失效，造成系统在开机后自动产生 IRQ7 中断的机器上启动失败，无限重启。
解决方案：已修正 gdtigt.c 中 initGdtIdt()函数的 GDT、IDT 地址范围数

据。无限重启漏洞已成功修复，在 qemu 虚拟机 bochs 虚拟机、VM 虚拟机上均测试正常。

发现者：年年软件；发现日期：2019.3.11；修复日期：2019.3.23。

4.3.4 缓冲区溢出漏洞

编号：KB00000003。状态：已修复。命名：尚未命名。出现版本：NNOS 0.16i。

描述：开机后 3 秒，系统失去响应。

原因分析：经进一步检查确认，开机后使用鼠标，字符光标定时器的缓区会被鼠标中断信号污染。在开发 0.16i 版本时发现此漏洞，转而检查 0.16h 版本时，发现此漏洞在 0.16h 上同样存在，只是表现形式不同。起初判定为定时器中断处理器存在逻辑错误，但在随后的分析中，发现停用字符光标定时器后问题消失，经过进一步调试分析，方才确定问题出在先进先出缓冲区的 put、get 算法，该算法存在算术错误，导致潜在的缓冲区溢出风险，此风险自 FIFO 缓冲区建立时便一直存在，在合并缓冲区和提高中断处理器速度之前未表现出来。0.16h 开始爆发，0.16h 的定时器中断处理器速度较慢，因此漏洞表现为定时器缓冲区污染，晃动鼠标时所有定时器停止工作。0.16i 中，定时器中断处理器的速度大大提高，因此漏洞表现为鼠标缓冲区污染，晃动鼠标数秒后，鼠标失去响应。

解决方案：已确认为 FIFO 缓冲区算法中算术错误导致的缓冲区溢出。已修正相应的错误算法，修正后恢复正常，测试良好。

发现者：年年软件；发现日期：2019.3.23；修复日期：2019.3.24。

4.3.5 多任务死机漏洞

编号：KB00000004。状态：已修复。命名：待命名。出现版本：NNOS 0.18f。

描述：开机后即死机，但计时器有记录，说明并未立刻死机。

原因分析：在关闭多任务模块后，系统运行正常，可以确定该问题为任务切换存在逻辑错误。经过对源代码的筛查和程序调试，发现此问题是由定时器中断处理器中多任务自动切换定时器标志信号声明范围错误引起的。该声明应为中断处理器的局部变量，但却错误地声明为全局变量，造成开机后从主任务向其他任务进行任务切换时，溢出的标志位无法满足切回条件，在当前任务停滞，且系统无法再处理其他中断，鼠标、键盘失去响应，系统死机。

解决方案：已确认为中断处理器多任务定时器标志信号声明范围错误，已由全局变量修正为局部变量。修复后测试正常。

发现者：年年软件；发现日期：2019.3.31；修复日期：2019.3.31。

4.3.6 文件信息区内存污染漏洞

编号：KB00000005。状态：已修复。命名：待命名。出现版本：NNOS

0.22a。

描述：控制台“type”'(或“cat”)命令异常，查看文件内容后，文件拓展名被非法写入磁盘文件信息区的所有文件拓展名前。

原因分析：在查询时，定义了 fileName 数组，该数组之内存空间未经操作系统管理，导致该空间与文件系统文件信息区文件拓展所用内存地址相同。'type'命令执行时，需要将用户输入转为代写，调用了 convertToUppercase()函数，该函数传入一个文件名字符数组首地址，对逐个字符进行操作，进行该操作的同时，污染了文件信息区的所有文件拓展名前三个字节，表现为用户输入的所查看文件的拓展名被写入所有文件的拓展名之前。经进一步分析，确定真正问题根源在于相关数组未初始化。

解决方案：初级解决方案，取消常规数组定义，使用字符指针，经过 NNOS 的内存管理系统申请可控内存空间，从而防止内存污染。终极解决方案，在数组定义时初始化置空值。

发现者：年年软件；发现日期：2019.4.4；修复日期：2019.4.4。

4.3.7 一般异常处理死循环

编号：KB00000006。状态：已修复。命名：待命名。出现版本：NNOS 0.24d。

描述：分离系统程序段与应用程序段并增加应用一般异常中断处理程序后，在控制台使用“run”运行可执行文件皆触发此异常并无限循环。

原因分析：经排查，IDT 注册无问题。中断处理函数和 run 命令无问题。最终确定问题出现在 osfun.asm 中 API 中断处理器_asm_sys_api 中，有一处寄存器地址数值书写错误，被扩大 10 倍。

解决方案：已修正上述数值错误，系统运行、程序运行测试正常。

发现者：年年软件；发现日期：2019.4.9；修复日期：2019.4.9。

五 开发环境搭建

5.1.1 开发平台及开发环境

所谓开发平台，即开发本操作系统时的用操作系统。本项目参考文件推荐的开发平台为 Windows,所提供的编译程序也全部是 Windows 版本。但是考虑到日后系统开源、升级维护的需要，笔者在开发前期花费了大量时间精力，完成了开发环境从 Windows 到 Linux 的兼容。为此笔者专门开发了源代码转换程序、各个版本的文件生成规则和适用于两个平台的一键编译脚本。详细内容将在后面的章节介绍。

5.1.2 编程语言

（1）机器语言。系统初期测试性程序，将直接使用机器语言。虽然繁琐，

但本质上只有机器语言才是计算机唯一可识别的语言，汇编语言和其他高级语言编写的源代码最终都要编译为二进制的机器语言程序方才能在机器上运行。使用机器语言编写测试程序，能够切身体会冯诺依曼原理的“指令和数据同等地位存储”，亦能切身和底层硬件打交道，对微机原理有更深的认识，为下一步开发扫清一定的障碍。

（2）汇编语言。本系统采用 ASM 汇编语言开发底层模块（IPL 引导扇区、实模式到保护模式的过渡程序）。汇编语言有硬件相关性，并且可以直接调用硬件终端指令。对于设计系统底层的功能实现，有机器语言与汇编语言两个选择，显然前者非人力所能为。因此，汇编语言是事实上的唯一选择。Intel 指令集参考手册对 80x86 汇编有着强大的支持，国内外技术网站也有大量参考资料可供学习、借鉴。从这一点来看，汇编语言也是底层模块开发的最佳选择。综合以上因素，系统的初期开发和后续设计硬件的函数实现采用汇编语言编写。

（3）C 语言。本系统后期非底层部分（主程序）采用 C 语言开发。C 语言是众多高级语言的鼻祖，简洁而稳定。最重要的是，C 语言的支持指针，可以通过指针方便地实现对内存地址的访问，而其他语言诸如 JAVA 虽有面向对象的优势，但操作系统开发上，基本上所有系统相关的函数库都是无法使用的，再加上没有指针的种种不便，使用 JAVA 绝非明智选择。综上，C 语言成为本次开发中的主力。

5.1.3 编译器

（1）汇编语言编译器。在 Windows 上，采用 nasm 的优化版 nask。这也是本文主要参考文献《30 天自制操作系统》所推荐的编译器，该编译器开发者为即此书作者川合秀实先生。在 Linux 上，前期采用 nasm，后期通过网络获取 nask 的 Linux 版本，改用 nask 编译。

（2）C 语言编译器。在 Windows 上，本系统所有 C 语言源文件均采用 gcc 4.9.2 进行调试，在 Linux 上则采用 gcc 7.3.0 版本。在编译生成目标程序时，由专用编译器完成源文件转换，这些编译器的作者，全部为川合秀实先生。

5.1.4 虚拟机

（1）采用虚拟机的原因。本系统的主要参考文献仅提供了对系统软盘镜像的的支持，笔者后期将着手实现系统的 U 盘启动。但是前期的主要精力还是应

集中在内核开发上，因此前期的测试仍然要依赖软盘进行。软盘的读写次数相当有限，并且极易损坏。而开发操作系统的过程中又需要频繁的软盘读写，消耗的软盘数量相当巨大，且目前的个人计算机早已不再内置软盘驱动器。笔者没有足够的资金购买开发所需要的软盘，更没有足够的资金购买开发所需要的软盘驱动器。因此只能退而求其次，使用虚拟机中的虚拟软盘来完成系统的调试、测试。

（2）主要测试用虚拟机。本系统开发过程中，涉及到的主要虚拟机有 Qemu、Bochs 和 VMware Workstation。其中 Qemu 为免费软件，Bochs 为开源软件，而 VMware Workstaion 为闭源软件。同样，由于资金问题，笔者无法购买昂贵的 VMware 正版授权软件，在开发过程中使用了 VM 虚拟机的破解版，不能不说是一个可耻的行为。笔者将来有了独立的经济来源，一定会购买支持正版软件。本系统不会被用于商业用途，望谅解。对于各个虚拟机在测试过程总发挥的左右，在测试分析一节中将做详细介绍。

4.1.5 文件生成规则

系统开发初期使用的编译、测试较少，逐条手动输入编译命令不成问题。但是随着开发的深入，有大量重复的编译、测试命令需要输入，逐条手动输入是低效的。Windows 下的批处理脚本文件和 Linux 下的 shell 脚本文件可以胜任此工作，但还不够完美。本系统的每次迭代，几乎都有编译命令的增改，使用脚本的话，需要手动修改命令，这个工作量和手动逐条输入命令也差不了多少。更重要的是，迭代过程中已经开发完成的源代码在短时间内（主版本号变更前）不会发生变化，链接过程中又需要所有目标文件，可是在开发过程中难以记忆、分辨哪些目标文件是更新过的，使用脚本程序完成编译命令的执行会只能重新编译已经编译过的源文件（如果要写分辨文件变化的程序，又是一个额外的巨大工程），这会白白耗费大量时间，拖累工程进度。

基于以上考虑，本系统开发过程中使用自由软件基金会（GNU）提供的 Makefile 文件生成规则来完成编译命令的智能执行。这大大节省了开发时间，在此对 GNU 表示感谢。

5.1 环境和工具列表

5.1.1 操作系统和硬件要求

①Windows 7 及以上操作系统或 GNU/Linux 的任意发行版；
②内存至少 2GB，硬盘可用空间不小于 2GB；
③四肢健全或能够熟练用脚打字。

5.1.2 工具软件列表

（括号内注明用途或适用的平台）
①nasm 编译器、nask 编译器（Winows/Linux）；
②gcc 编译器（Windows 下可先安装 Dev-c++然后配置环境变量，具体可百度）；
③VMware Workstation 虚拟机（Windows），也可用 Vitrul Box，不推荐；
④QEMU 虚拟机（Windows/Linux）、Bochs 虚拟机（Linux/Windows）；
⑤Make（文件生成规则解析器，Window/Linux）；
⑥NNOS 开发工具包（已包含在 Github 项目的源文件中，Windows/Linux）；
⑦NotePad++（Windows）、VSCode（Windows/Linux）；
⑧Hex Editor Neo（二进制/十六进制编辑器，Windows）
⑨vim（编辑器，Linux）、fish（终端，可选，linux）、git 客户端
⑩Chrome 浏览器（可选，WInows/Linux）、极速 PDF 阅读器（可选）

5.2 工具简介

5.2.1 开发环境搭建

为完成操作系统开发，需要额外准备通用开发环境，以方便开发和测试。

（1）Windows 平台下的通用开发环境。若计算机已经安装 Dev-c++集成开发环境和 VSCode 编辑器，VSCode 编辑器的 C/C++插件、Dev-c++根目录的 MinGW64 文件夹中，都内置了 gcc 编译器，因此无需额外安装 gcc，只需要将 gcc 所在目录在系统环境变量的 path 中注册即可。而 bochs 虚拟机，则需要从 Bochs 官网下载。和大多数开源软件一样，Bochs 提供安装版和解压版，考虑到稳定性，推荐使用 Bochs2.6.9 的解压版，直接在软件安装目录解压，然后将 Bochs 主程序所在目录在系统公用环境变量的 path 表项中注册即可。Qemu 虚拟机安装

与 Bochs 类似，考虑到稳定性，不推荐使用最新版本。Windows 平台下的汇编语言环境，需下载安装 nasm 2.14 的解压版，在环境变量中注册。此外，可从开发工具包（tools，下一节将作详细说明）中提取 nask，并将放入软件安装目录，在环境变量中注册。实际开发过程中，使用 nask 进行编译时，Makefile 的默认配置所使用的 nask 仍然是开发工具包中的 nask。工具包中的 nask 在 Windows 命令行下只能使用绝对路径或相对路径来调用，但是编译出现问题需要调试时，不可能手动输入开发工具包的相对路径来调用 nask。直接将工具包中的 nask 在环境变量中注册也是不现实的，因为工具包会跟随项目的迭代更改位置[1]，为避免潜在的风险，只能准备两个 nask，一个在开发工具包中，一个在系统软件安装目录并注册。此外 Make.exe 亦须注册。

（2）Linux 平台下的通用开发环境。在 VM 虚拟机中安装完 Linux 系统的发行版 Ubuntu18.04 后，需要通过终端命令安装与 Window 下相对应的编译器、测试用虚拟机的 Linux 版。安装过程中使用的主要命令（部分命令从略）如表 5.1.3 所示。

表 5.1.3 Linux 通用开发环境所涉及的命令

命令	参数	功能	备注
sudo apt-get updata	无	检查更新	安装任何软件前须执行
sudo apt-get upgrade	无	安装更新	须在上一条之后执行
sudo apt-get install	gcc-4.9[注]	安装 gcc4.9	本表难容全部参数，见表注
sudo apt-get install	qemu	安装 qemu 虚拟机	实际安装后需要安装依赖
tax -zxvf	bochs-2.6.9.tar.gz -C	解压 bochs 虚拟机	实际解压后需要手动配置
tax -zxvf	Nasm-2.14.03rc2.tar.gz	解压 nasm 编译器	实际解压后需要手动配置

[1] 虽然工具包中的编译工具基本不会发生变化，大可以一直使用某个旧版本目录下的 nask 作为默认 nask，但开发操作系统的过程中，任何一个细微处稍有不慎，便会节外生枝、焦头烂额，这一点笔者是有深刻教训的

[注]gcc 安装的命令比表中要负责，需要安装多个版本的 gcc，并更改配置文件，使之可以通过命令切换。安装完成后，须切换为 4.9 版本。

此外 Linux 下还需要安装开发工具包，川合秀实先生所提供的工具包仅支持 Windows 版本，笔者起初为了兼容，使用了 Linux 其他具有相同功能的软件。比如镜像制作工具，使用 Bochs 内置的 bximage 命令，加上管道传参来完成；软盘数据写入，则使用 dd[1]命令写入；而 nask 则使用 nasm 代替。但是后期多个目标文件链接时，非指定工具包无法完成。笔者几乎要放弃对 Linux 开发环境的兼容了，幸运的是借助互联网，笔者找到了图灵社区所发布的开发工具包的 Linux 版本，这么一来，Windows 下的所有专用编译工具在 Linux 下也可以使用了。这些工具无需繁琐的配置，笔者直接将它与 Window 下的开发工具包放在同一主文件夹下，然后在 Linux 版的 Makefile 中指定此路径，开发工具包就可以使用了。

5.2.4 开发工具包详情

开发操作系统的过程中，涉及到汇编语言和 C 语言的混编，涉及到多种目标文件的链接，这些工作 nasm 编译器和 gcc 编译器都无法独立完成，需要专用的开发工具包。此开发工具中的工具除 make.exe 外均由川合秀实先生开发并提供。先生提供了源代码，所有工具均具有开源许可。原目录名称为 z_tools，笔者对其中的依赖文件进行了适应性改进（后文将详细介绍），更名为 tools。而 Linux 版本中每个开发工具都与 tools 中的开发工具相对应，但是两个文件夹要放在同一主目录下，笔者用更改的依赖文件替换原目录中的文件后，将 Linux 下的开发工具包更名为 tools_linux。两个文件夹中的开发工具名称完全相同，功能也完全相同。所有开发工具、依赖文件及其功能如表 5.1.4 所示。

表 5.1.4 开发工具包工具列表

名称	功能	授权
nask.exe	编译、调试 nask 汇编源文件	开源许可协议
edimg.exe	软盘镜像生成、数据写入	开源许可协议
imgtol.exe	物理软盘数据写入	开源许可协议

[1] 此工具也存在 Windows 版，在开发初期，Windows 下也可使用 dd 和 winimage、FloppyWrite 完成软盘镜像制作。

ccl.exe	将 C 语言转换为 gas 过渡文件	开源许可协议
gas2nask.exe	将 gas 过渡文件转换为 nask 源文件	开源许可协议
obj2bim.exe	将 obj 目标文件转换为 bim 文件	开源许可协议
eimtohrb.exe	将 bim 文件转换为二进制文件	开源许可协议
makefont.exe	将点阵字库转换为二进制文件	开源许可协议
bin2obj.exe	将二进制文件转换为 obj 目标文件	开源许可协议
make.exe	文件生成规则解析	开源许可协议

[注]Linux 下的开发工具包中不包含 make.exe，因为在 Linux 环境下，Makefile 文件生成规则解析工具 make 作为 GNU 的一员，默认被集成在了部分发行版本中。即使没有继承，亦可使用终端进行安装。

有了以上开发工具，可以顺利地实现汇编语言与 C 语言的混编、链接以及软盘镜像写入，在源文件编译上节省了不少力气。但是目前主流的汇编编译器为 nasm，而参考文献指定的汇编编译器为它的优化版版 nask。笔者经分析发现，nasm 与 nask 仅有少量命令存在区别，于是编写了转换程序 getnas，来讲.asm 源文件转换为.nas 源文件。此程序兼容 Windows 和 Linux，将在后文详细说明。

对主要开发工具的介绍到此告一段落，更多细节在后文会提及。

5.2.5 编辑器和阅读器

好的编辑器可以使得编程过程顺畅舒适，编辑器提供的各种插件（比如代码提示、文本比较）等，可以减少重复劳动、节省时间精力。

（1）二进制编辑器。开发操作系统的早期实验阶段，涉及到机器语言，需要直接编写二进制文件（实际编辑器为十六进制，因为十六进制可以直观地完成与二进制的转换，而比二进制更简洁），因此需要准备二进制编辑器。参考文献所推荐的 BZ,年代久远，异常难用。笔者使用了它的后辈，最新版本的 Hex Editor Neo 来代替。在 Linux 下，可以直接 vim，在命令模式下输入“%xxd0”即可编辑修改二进制文件。

（2）汇编、C 语言编辑器。系统早期使用汇编语言，后期主体使用 C 语言。选用的编辑器为久负盛名的 Notepad++，版本为 v7.6.4，并安装了文本比较插件。在开发过程中，笔者还先后尝试过 Sublime Text3 和 VSCode，尽管它们能够提供更多的代码提示、函数声明与实现跳转等功能，但本系统所涉及的源文件太多，

这两个编辑器不够简洁，无法让人集中精力于代码本身。笔者切身体会，使用Notepad++读代码、改 BUG 的效率更高，也更容易发现潜在风险，是故最终决定使用它完成后续全部开发。至于 Eclipse、Dev-C++乃至 Visual Studio 等集成开发环境，他们的强大大多只能使用在开发应用软件上，对于操作系统这样的系统软件则显得臃肿而格格不入，何况开发过程中使用大量脚本，直接在终端运行这些脚本可以比在图形界面上点击鼠标更方便快捷地完成编译、调试。在 Linux 下没有 Notepad++，终端使用 fish，编辑器使用 vim，已经足矣。

（3）阅读器。笔者在开发过程中，需要阅读大量参考文献，这些文献大都有电子版[1],阅读这些电子版文献，一个好的阅读器太重要了，笔者尝试过 Chrome 浏览器、Edge 浏览器直接打开，自然方便，但许多笔者需要的功能都没有。而经典的方正阿帕比(Apabi Reader)在全屏阅读和批注上令人抓狂。最终，笔者选择了极速 PDF 阅读器。而对于知网论文，则不得不用 CAJViewer 阅读器了。

[1] 说来惭愧，这些书籍许多没有正版授权，但是以笔者的经济水平，实在无力购买如此多的参考书籍实体书，笔者连饭钱也已经交了房租了(笑)。笔者有了经济实力，一定会购买这些书籍。

附录

附录 | 硬件相关规定

表 1 通用寄存器及其用途规定

汇编助记符	惯用名称	用途（规定）	备注
AX	Accumulator(累加寄存器)	通用，作累加器	可拆为 AH 和 AL
DX	Data（数据寄存器）	通用，常存数据、	可拆为 DH 和 DL
CX	Countor（计数寄存器）	通用，作计数器	可拆为 CH 和 CL
BX	Base（基址寄存器）	通用，偏移	可拆，常与 DS 连用
SI	Source Index（源变址寄存器）	通用，源地址	串数据访问
DI	Destination Index	通用，目的地址	串数据访问
SP	Stack pointer（栈指针寄存器）	存放栈顶地址	用于访问堆栈数据
BP	Base pointer（基址指针寄存器）	存放栈帧地址	用于访问堆栈数据

[注]在 32 位寄存器中与 16 位寄存器共用低 16 位地址，相应助记符前加“E”。表中名均略去“Register”。

表 2 段寄存器及其用途规定

汇编助记符	惯用名称	用途（规定）	备注
ES	Extra Segment（附加段寄存器）	存放附加段段基地址	段寄存器
DS	Data Segment（数据段寄存器）	存放数据段基地址	段寄存器
CS	Code Segment（代码段寄存器）	存放代码段基地址	段寄存器
SS	Stack Segment（栈寄存器）	存放堆栈段基地址	段寄存器
IP	Instruction Pointer（指令指针寄存器）	存放前须取的指令	专用，不可直接操作
FLAGS	状态标志寄存器。	存放处理器状态信息	专用寄存器

[注]关于寄存器的详细介绍，请参看任意 Intelx86 处理器软件开发手册。

表 3 软盘读写 BIOS 指令及标志信息

指令/指令模板/标志位信息	功能	备注
AH = 0x02	读盘	
AH = 0x03	写盘	

AH = 0x0c	寻道	
AL = 扇区数	指明待处理的扇区数	只能为连续扇区
CH = 柱面号 & 0xff	指明柱面号	
DH = 磁头号	指明磁头号	

续表 11

指令/指令模板/标志位信息	功能	备注
CL = 扇区号	指明扇区号	
DL = 驱动器号	指明驱动器号	驱动器号，即卷号
ES:BX = 缓冲区地址	指明缓冲区首地址	在校验、寻道时不使用
FLAGS.CF == 0	进位标志为 0，读盘未出错	
FLAGS.CF == 1	进位标志为 1，读盘出错	错误码将写入寄存器 AH

表 4 VGA 显卡设定相关 BIOS 指令

指令/指令模板	功能	备注
AH = 0x00	指明显卡模式	
AL = 0x03	设定为 16 位彩色画面模式	
AL = 0x12	设定为 640×480×4 位彩色画面模式	
AL = 0x13	设定为 320×200×8 位彩色画面模式	
AL = 0x	设定为 800×600×4 位彩色画面模式	部分显卡不支持

[注]非 VBE 用“AH=0”和“AL=画面模式号码”指定。VB 用“AX=0x4f02”和“BX=画面模式号码”指定。

表 5 VBE 显卡设定相关 BIOS 指令

指令/指令模板	功能	备注
BX = 0x101	指明 640×480×8 位彩色画面模式	
BX = 01x03	指明 800×600×8 位彩色画面模式	
BX = 0x105	指明 1024×768×8 位彩色画面模式	
BX = 0x107	指明 1280×1024×8 位彩色画面模式	本系统不支持

表 6 VBE 画面模式信息所在位置

地址	信息
[ES:DI+0x00]	模式的属性，第 7 比特位须为 1
[ES:DI+0x12]	屏幕横向分辨率

[ES:DI+0x14]	屏幕纵向分辨率
[ES:DI+0x19]	屏幕颜色位数，须为 8 位
[ES:DI+0x1b]	颜色指定方法，须为 4（调色板模式）
[ES:DI+0x28]	VRAM 首地址

附录 II 项目源文件列表

表 7 系统源文件列表及其主要作用

编号	文件名	层次	模块	功能简介	备注
01	ipl.asm	引导层	IPL	引导程序，引导系统启动。. nas 同。	汇编语言
02	syshead.asm	引导层	SYSHEAD	系统实模式程序	汇编语言
03	osfun.asm	引导层	OSFUN	底层函数库，C 下无法完成的实现	汇编语言
04	nnos.h	基础层	Manager Interface	头文件，系统函数声明	C 语言
05	syshead.h	基础层	Manager Interface	系统结构体定义	C 语言
06	systructual.h	基础层	Manager Interface	系统常量定义、静态数组定义	C 语言
07	bootpack.c	核心层	BOOT	主程序，保护模式下运行	C 语言
08	gdtidt.c	基础层	GDT&IDT	GDT 和 IDT 相关函数实现	C 语言
09	interrupt.c	基础层	INTERRUPT	中断相关函数实现	C 语言
10	timer.c	基础层	TIMER	定时器相关函数实现	C 语言
11	fifo.c	基础层	FIFO	先进先出缓冲区相关函数实现	C 语言
12	memery.c	核心层	Memery Manager	内存管理函数实现	C 语言
13	devices.c	核心层	I/O Manager	设备管理函数实现	C 语言
14	task.c	核心层	Process Manger	进程管理和多任务函数实现	C 语言
15	file.c	核心层	File Manger	文件管理函数实现	C 语言
16	graphics.c	核心层	Graphics	图形绘制引擎、图形界面函数实现	C 语言
17	fontbase.frc	扩展层	Font Base	基本点阵字体文件	字体文件
18	cover.c	扩展层	Layer Drawer	图层处理引擎	C 语言
19	console.c	应用层	Console	系统控制台应用程序	C 语言
20	hello.nex	应用层	Other App	汇编语言应用程序示例	应用
21	hello02.nex	应用层	Other App	C 语言应用程序示例	应用

22	hello03.nex	应用层	Other App	C 语言调用 API 应用程序示例	应用
23	update.log	/	/	系统升级日志	日志
24	buglist.log	/	/	系统内核漏洞列表	日志
25	getnas.c	/	/	nasm 兼容 nask（Windows 版）	C 语言
26	getnas_linux.c	/	/	nasm 兼容 nask 转换器（Linux 版）	C 语言
27	Makefile	/	/	文件生成规则，编译自动化	编译用脚本

续表 15

编号	文件名	层次	模块	功能简介	备注
28	Makefile_win	/	/	文件生成规则，兼容 Windows 平台	编译用脚本
29	Makefile_linux	/	/	文件生成规则，兼容 Linux 下开发	编译用脚本
30	run_win.bat	/	/	一键编译运行脚本（Windows 版）	编译用脚本
31	run_linux.sh	/	/	一键编译运行脚本（Linux 版）	编译用脚本
32	runpowershell.bat	/	/	powershell 启动脚本，仅 Windows	编译用脚本
33	nnos.rul	/	/	编译器依赖	二进制文件
34	bochsrc.bxrc	/	/	bochs 配置文件（Windows 版）	编译用脚本
35	bochsrc	/	/	bochs 配置文件（Linux 版）	编译用脚本
36	bochsout.txt	/	/	bohcs 虚拟机信息输出文件	普通文本
37	LICENSE	/	/	996ICU 开源协议	协议
38	nnos.img	/	/	编译链接后最终生成的系统软盘镜像	二进制文件

［注］本表据 nnos0.24d 源文件目录建立，对其他版本部分适用。以上文件可能随系统迭代变化，这些变化包括但不限于增加、删除、合并、拆解、移动位置或重命名，本表也会随系统迭代进行更新。

附录 III 系统常量表

表 1 系统属性常量

键	值	含义	备注
NNOS_VERSION	“NNOS 0.18a”	系统版本号	随迭代更新

表 17 系统信息常量

键	值	含义	备注
NNOS_FILE_SYSTEM	"FAT12"	文件系统格式	FAT12 格式
NNOS_CPU_BASED	"Intel_80x86"	CPU 架构	
NNOS_COMPANY	"NNRJ"	工作室	年年软件
NNOS_AUTHOR	"Liu Dongxu"	作者	刘东旭
NNOS_EMAIL	"tianhehechu@qq.com"	电子邮箱	开发者邮箱
NNOS_UPDATE_DATE	"2019-4-7"	更新日期	

表 2 BOOT 信息首地址常量

键	值	含义	备注
BOOTINFO_ADR	0x00000ff0	BOOT 信息内存首地址	

表 19 显卡信息常量

键	值	含义	备注
VWIDTH	320	屏幕宽度，单位为像素	引入 BOOT 信息后已废弃
VHEIGHT	200	屏幕高度	引入 BOOT 信息后已废弃
COLORNUM	26	颜色表大小（颜色数目）	用于遍历颜色表

表 3 GDT 常量

键	值	含义	备注
GDT_ADR	0x00270000	GDT 基地址	
GDT_LIMIT	0x0000ffff	GDT 上限	
DATA32_RW_PRE	0X4092	应用模式，读写	同适用于 IDT

CODE32_ER_PRE	0x409a	系统模式，运行	同适用于 IDT
BOOT_ADR	0x00280000	BOOT 首地址	
BOOT_LIMIT	0X0007ffff	BOOT 地址上限	
GDT_SELECTOR_MAX	8192	GDT 选择子最大值	

［注］本表受 x86 处理器制约。

表 4 IDT 常量

键	值	含义	备注
IDT_ADR	0x0026f800	IDT 基地址	
IDT_LIMIT	0x000007ff	IDT 上限	
INTGATE32_PRE	0x008e	允许中断	
TSS32_PRE	0x0089	多任务权限	

表 5 PIC 相关常量

键	值	含义	备注
PIC0_ICW1	0x0020		
PIC0_ICW2	0x0021		
PIC0_ICW3	0x0021		
PIC0_ICW4	0x0021		
PIC0_OCW2	0x0020		
PIC0_IMR	0x0021		
PIC1_ICW1	0x00a0		
PIC1_ICW2	0x00a1		
PIC1_ICW3	0x00a1		
PIC1_ICW4	0x00a1		
PIC1_OCW2	0x00a0		
PIC1_IMR	0x00a1		

表 6 PIT 相关常量

键	值	含义	备注
PIT_CTRL	0x0043	PIT 端口号	
PIT_CNT0	0x0040	PIT 端口号	

TIMER_MAX	500	定时器最大数量
TIMER_FLAG_UNUSED	0	定时器未使用
TIMER_FLAG_USED	1	定时器已使用
TIMER_FLAG_USING	2	定时器正在运行
TIMER_TIMER_MAX	42949673	定时器溢出上限(十进制)
TIMER_TIME_MAX_H	0xffffffff	定时器溢出上限(十六进制)

表 7 内存描述相关常量

键	值	含义	备注
MEMERY_ADDR	0x003c0000	内存首地址	
MEMERY_LIST_SIZE	4096	内存空闲块表大小	单位为“块”
SYS_PRE	0x00400000	系统预留内存	4MB
MEMERY_MAX_SIZE	0x07c00000	支持的最大内存容量	

表 8 内存处理常量

键	值	含义	备注
EFLAGS_AC_BITS	0x00040000	CPU 类型校验码	486 及以上有 AC 位
CR0_CACHE_DISABLE	0x60000000	禁用 Cache 操作码	
BLOCK_ALIGN	0xfffff000	内存块 4K 对齐操作数	4K=0x1000，已废弃

[注]本表前两项为对 CPU 操作时使用的常量。

表 9 文件信息常量

键	值	含义	备注
DISK_ADR	0x00100000	文件系统磁盘首址	
FILE_INFO_MAX	224	文件信息最大个数	受 FAT12 限制
FILE_NAME_SIZE	8	文件名最大长度	受 FAT12 限制
FILE_EXTEN_SIZE	3	拓展名最大长度	受 FAT12 限制

FILE_FULL_NAME_SIZE	FILE_NAME_SIZE+FILE_EXTEN_SIZE	文件全名最大长度

表 10 鼠标指针相关常量

键	值	含义	备注
CURSOR_WIDTH	16	鼠标指针宽度	
CURSOR_HEIGHT	16	鼠标指针高度	
CURSOR_PIX	16	鼠标指针分辨率	即将废弃
CURSOR_DATA_SIZE	3	鼠标指针数据大小	即将废弃
CURSOR_BCOLOR	DESK_BCOLOR	鼠标指针背景色	
CURSOR_GRAPH_SIZE	CURSOR_WIDTH * CURSOR_HEIGHT	鼠标指针位数	16×16

表 11 鼠标处理相关常量

键	值	含义	备注
KEYCMD_MOUSE	0xd4	键盘控制电路切鼠标模式	
MOUSECMD_ENABLE	0xf4	鼠标激活指令	
ACK	0xfa	鼠标激活成功后确认信息	
MOUSE_DATA_BASE	512	鼠标中断信号偏移量	

表 12 键盘端口相关常量

键	值	含义	备注
PORT_KEYDAT	0x0060	键盘设备端口号	
PORT_KEYSTA	0x0064	键盘状态端口号	
PORT_KEYCMD	0x0064	键盘控制电路端口号	
KEYSTA_NOTREADY	0x02	键盘未准备好	
KEYCMD_WRITE_MODE	0x60	键盘控制电路模式指令	
KBC_MODE_MOUSE	0x47	键盘控制电路鼠标模式	
KEY_DATA_BASE	256	键盘中断信号偏移量	

表 13 FIFO 缓冲区相关常量

键	值	含义	备注
FLAGS_OVERRUN	0x0001	缓冲区溢出	
KEY_BUFFER_SIZE	32	键盘缓冲区大小	已废弃
MOUSE_BUFFER_SIZE	128	鼠标缓冲区大小	已废弃
TIMER_BUFFER_SIZE	8	定时器缓冲区大小	已废弃
BUFFER_SIZE	128	通用缓冲区大小	

表 14 桌面相关常量

键	值	含义	备注
DESK_WIDTH	VWIDTH	桌面默认宽度	已废弃
DESK_HEIGHT	VHEIGHT	桌面默认高度	已废弃
TASKBAR_WIDTH	DESKWIDTH	任务栏默认宽度	已废弃

TASKBAR_HEIGHT	DESKHEIGHT*0.05	任务栏默认高度	已废弃
DESK_BCOLOR	COL8_004276	桌面默认背景色	

[注]“已废弃”的常量并非不再使用。

表 15 图层相关常量

键	值	含义	备注
COVER_LIST_SIZE	256	图层列表最大图层数	已废弃
COVER_UNUSE	0	图层未使用	
COVER_USEED	1	图层已使用	
COVER_NUM	15	图层最大图形数	已废弃

表 16 系统默认窗口定义常量常量

键	值	含义	备注
WINDOW_X	5	默认窗口横坐标	单位：像素。下同
WINDOW_Y	5	默认窗口纵坐标	
WINDOW_WIDTH	70	默认窗口宽度	
WINDOW_HEIGHT	50	默认窗口高度	
WINDOW_LINE_WIDTH	1	默认轮廓线宽度	
WINDOW_CAPTION_HEIGHT	20	默认窗口标题高度	
WINDOW_LINE_COLOR	COL8_101010	默认窗口轮廓线颜色	
CONTROL_WINDOW	1	控制台窗口	
NOMAL_WINDOW	2	普通窗口	
INFO_WINDOW	3	信息提示窗口	
TEXT_WINDOW	4	文本输入窗口	
CONSOLE_WIDTH	400	控制台默认宽度	
CONSOLE_HEIGHT	250	控制台默认高度	
CONSOLE_FORECOLOR	COL8_38CE2F	控制台默认前景色	
REFRESH_ALL	0	窗口区域全部刷新操作数	
REFRESH_CAPTION	1	窗口标题刷新操作数	
CMD_CURSOR_X	16	控制台默认光标横坐标	
CMD_CURSOR_Y	28	控制台默认光标纵坐标	

表 17 字体相关常量

键	值	含义	备注
FONT_SIZE	16	字符位数	
FONT_CHAR	1	字符字节数	

表 18 颜色名常量

键	值	含义	备注
COL8_000000	0	纯黑	
COL8_FF0000	1	纯红	
COL8_00FF00	2	纯绿	
COL8_0000FF	3	纯蓝	
COL8_FFFF00	4	纯黄	
COL8_FF00FF	5	纯紫	
COL8_00FFFF	6	纯青	
COL8_FFFFFF	7	纯白	
COL8_C6C6C6	8	纯灰	
COL8_840000	9	暗红	
COL8_008400	10	暗绿	
COL8_000084	11	靛青	
COL8_848400	12	暗黄	
COL8_848484	13	暗紫	
COL8_008484	14	靛蓝	
COL8_848484	15	暗灰	
COL8_005B9E	16	湛蓝	
COL8_0078D7	17	浅蓝	
COL8_004276	18	深蓝	
COL8_FFFFFE	19	墨白	
COL8_E1E1E1	20	浅灰	
COL8_101010	21	明黑	
COL8_333333	22	黑灰	
COL8_D9D9D9	23	银灰	
COL8_E81123	24	亮红	
COL8_F0F0F0	25	薄灰	

表 19 系统颜色别名常量

键	值	含义	备注
FRESH_BLUE	COL8_005B9E	湛蓝	
TEEN_BLUE	COL8_0078D7	浅蓝	
DEEP_BLUE	COL8_004276	深蓝	
FLOUR_WHITE	COL8_FFFFFE	墨白	
REAL_RED	COL8_FF0000	纯红	
REAL_YELLOW	COL8_FFFF00	纯黄	
REAL_BLUE	COL8_0000FF	纯蓝	
REAL_GREEN	COL8_00FF00	纯绿	

表 20 其他公用常量

键	值	含义	备注
ZERO_ADR	0x00000000	特殊地址，零地址	
FULL_ADR	0xffffffff	特殊地质，最大地址	
NULL	0	空	与 C 语言冲突，已废弃
TRUE	1	真	
FALSE	0	假	
NONE	-1	无结果	

附录Ⅳ 系统颜色信息注册表

表 1 系统颜色信息注册表

颜色序号	Red	Green	Blue	颜色名	颜色别名	备注
0	0x00	0x00	0x00	COL8_000000	/	纯黑
1	0xff	0x00	0x00	COL8_FF0000	REAL_RED	纯红
2	0x00	0xff	0x00	COL8_00FF00	REAL_GREEN	纯绿
3	0x00	0x00	0xff	COL8_0000FF	REAL_BLUE	纯蓝
4	0xff	0xff	0x00	COL8_FFFF00	REAL_YELLOW	纯黄
5	0xff	0x00	0xff	COL8_FF00FF	/	纯紫
6	0x00	0xff	0xff	COL8_00FFFF	/	纯青
7	0xff	0xff	0xff	COL8_FFFFFF	/	纯白
8	0xc6	0xc6	0xc6	COL8_C6C6C6	/	纯灰
9	0x84	0x00	0x00	COL8_840000	/	暗红
10	0x00	0x84	0x00	COL8_008400	/	暗绿
11	0x00	0x00	0x84	COL8_000084	/	靛青
12	0x84	0x84	0x00	COL8_848400	/	暗黄
13	0x84	0x00	0x84	COL8_840084	/	暗紫
14	0x00	0x84	0x84	COL8_008484	/	靛蓝
15	0x84	0x84	0x84	COL8_848484	/	暗灰
16	0x00	0x5b	0x9e	COL8_005B9E	FRESH_BLUE	湛蓝
17	0x00	0x78	0xd7	COL8_0078D7	TEEN_BLUE	浅蓝
18	0x00	0x42	0x76	COL8_004276	DEEP_BLUE	深蓝
19	0xff	0xff	0xfe	COL8_FFFFFE	FLOUR_WHITE	墨白
20	0xe1	0xe1	0xe1	COL8_E1E1E1	/	浅灰
21	0x10	0x10	0x10	COL8_000000	/	明黑
22	0x33	0x33	0x33	COL8_000000	/	黑灰
23	0xd9	0xd9	0xd9	COL8_000000	/	银灰
24	0xe8	0x11	0x23	COL8_000000	/	亮红

附录 V 键盘字符映射表

表 2 键盘字符映射表

序号	值	字符	ASCII	序号	值	字符	ASCII	序号	值	字符	ASCII	序号	值	字符	ASCII
0		空	00	27	]	]	1B	54		右 Shift	36	81	3	小 3	51
1		ESC	01	28		Enter	1C	55	*	小*	37	82	0	小 0	52
2	1	1	02	29		左 Ctrl	1D	56		左 Alt	38	83	.	小.	53
3	2	2	03	30	A	A	1E	57		Space	39				
4	3	3	04	31	S	S	1F	58		CapsLock	3A				
5	4	4	05	32	D	D	20	59		F1	3B				
6	5	5	06	33	F	F	21	60		F2	3C				
7	6	6	07	34	G	G	22	61		F3	3D				
8	7	7	08	35	H	H	23	62		F4	3E				
9	8	8	09	36	J	J	24	63		F5	3F				
10	9	9	0A	37	K	K	25	64		F6	40				
11	0	0	0B	38	L	L	26	65		F7	41				
12	-	-	0C	39	;	;	27	66		F8	42				
13	=	=	0D	40	‘	‘	28	67		F9	43				
14		退格	0E	41	`	``	29	68		F10	44				
15		TAB	0F	42		左 Shift	2A	69		Numlock	45				
16	Q	Q	10	43	\	\	2B	70		ScrollLock	46				
17	W	W	11	44	Z	Z	2C	71	7	小 7	47				
18	E	E	12	45	X	X	2D	72	8	小 8	48				
19	R	R	13	46	C	C	2E	73	9	小 9	49				
20	T	T	14	47	V	V	2F	74	-	小-	4A				
21	Y	Y	15	48	B	B	30	75	4	小 4	4B				
22	U	U	16	49	N	N	31	76	5	小 5	4C				
23	I	I	17	50	M	M	32	77	6	小 6	4D				

24	O	O	18	51	,	,	33	78	+	小+	4E
25	P	P	19	52	.	.	34	79	1	小 1	4F
26	[	[	1A	53	/	/	35	80	2	小 2	50

附录Ⅵ 系统函数简表

表 3 osfun.asm 所有函数及其功能

函数名	功能
_io_hlt	CPU 停机等待，相当于 void io_hlt();
_io_cli	屏蔽中断，相当于 void io_cli();
_io_sti	允许中断，相当于 void io_sti();
_io_stihlt	CPU 允许中断并停机等待，相当于 void io_stihlt();
_io_in8	从 I/0 端口读数据，8 位，指明端口号相当于 void io_in8(int port);
_io_in16	从 I/0 端口读数据，16 位，相当于 void io_in16(int port);
_io_in32	从 I/0 端口读数据，32 位，相当于 void io_in32(int port);
_io_out8	向 I/0 端口写数据，8 位，相当于 void io_in8(int port,int data);
_io_out16	向 I/0 端口写数据，16 位，相当于 void io_in16(int port,int data);
_io_out32	向 I/0 端口写数据，32 位，相当于 void io_in32(int port,int data);
_io_store_eflags	保存信息到 eflags，相当于 void io_store_eflgs();
_io_load_eflags	加载 eflags 信息，相当于 void load_eflags(int eflags);
_load_gdtr	加载 gdtr，相当于 void load_gdtr(int limit,int addr);
_asm_inthandler21	响应键盘中断，保护现场、调用键盘中断处理器、恢复现场
_asm_inthandler27	响应某些机器开机自用生成的中断，避免系统启动失败
_asm_inthandler2c	响应鼠标中断，保护现场、调用鼠标中断处理器、恢复现场

_asm_inthandler20	响应定时器中断，保护现场、调用定时器中断处理器、恢复现场
_load_cr0	加载 cr0 信息，相当于 int load_cr0();
_store_cr0	保存 cr0 信息，相当于 void store_cr0(int cr0);
_memtest_sub	内存检查子函数的汇编语言实现，绕过 C 语言编译器的优化
mts_loop	/
mts_fin	/

[注]本表中函数，完全参照川合秀实先生的 naskfun 编写。

表 4 gdt. c 函数列表及其功能

函数名	功能	备注
initGdtIdt()	初始化 GDT 和 IDT	参数略，下同
setGateDesc()	设定 GDT	
setGateDesc()	设定 IDT	

表 42 interrupt. c 函数列表及其功能

函数名	功能	备注
initPIC()	初始化 PIC	参数略，下同
inthandler21()	PS/2 键盘中断处理器	
inthandler2c()	PS/2 鼠标中断处理器	
inthandler27()	针对 PIC0 的不完全中断补丁	
initPIT()	初始化 PIT	
inthandler20()	定时器中断处理器	

表 5 fifo. c 函数列表及其功能

函数名	功能	备注
initFIFOBuffer()	初始化专用缓冲区	参数略，下同
putFIFOBuffer()	向专用缓冲区写入数据	
getFIFOBuffer()	从专用缓冲区读取数据	
getFIFOBufferStatus()	获取专用缓冲区数据状态	
initFIFOBuffer32()	初始化通用缓冲区	
putFIFOBuffer32()	向通用缓冲区写入数据	
getFIFOBuffer32()	从通用缓冲区读取数据	
getFIFOBufferStatus32()	获取通用缓冲区数据状态	

表 6 memery. c 函数列表及其功能

函数名	功能	备注
memeryCheck()	内存检查	参数略，下同
memtest_sub()	内存检查子函数	
initMemeryManager()	初始化内存管理系统	
memeryTotal()	内存余量统计	

memeryAlloc()	内存分配

续表 44

函数名	功能	备注
memeryFree()	内存释放	
deleteBlock()	移除内存块	
addBlock()	增加内存块	
sortMemery()	内存块按大小递增排序	
sortAddr()	内存块按地址递增排序	
memeryAlloc4k()	内存分配（4K 舍入）	
memeryFree4k()	内存释放（4K 舍入）	
clearMemery()	清理内存	

表 7 graphics. c **函数列表及其功能**

函数名	功能	备注
initPalette()	初始化画板	参数略，下同
setPalette()	画板设定	
pointDraw8()	像素点绘制	
boxDraw8()	色块绘制	
boxDrawx()	色块批量绘制	
initDesk()	初始化桌面	
fontDraw8()	字体渲染	
wordsDraw8()	字符串渲染	
initMouseCursor8()	绘制鼠标指针图像	
pictureDraw8()	图形绘制引擎	
windowDraw8()	窗口绘制引擎	
createWindow()	快速窗口创建	
labelDraw()	标签绘制	
makeTextBox8()	文本框绘制	
refreshWindowCaption()	窗口标题栏刷新	

表 8 devices. c 函数列表及其功能

函数名	功能	备注
initScreen()	初始化显示器	参数略，下同
initKeyboardCMD()	初始化键盘控制电路	
waitKBCReady()	等待键盘控制电路准备完毕	
enableMouse32()	激活鼠标	
decodeMouse()	鼠标解码引擎	
initKeyboardCMD32()	初始化键盘控制电路（32 位）	
sysHLT()	CPU 停机等待	

表 9 cover. c 函数列表及其功能

函数名	功能	备注
initCoverList()	初始化图层列表	参数略，下同
coverAlloc()	图层分配	
setCover()	图层设定	
updateCover()	图层更新	
coverRefresh()	图层刷新	
coverMove()	移除图层	
coverFree()	释放图层	

表 10 timer. c 函数列表及其功能

函数名	功能	备注
initTimer()	初始化定时器	参数略，下同
timerAlloc()	定时器分配	
setTimer()	定时器设定	
freeTimer()	定时器释放	
setTest480()	测试用定时器批量创建	无参。创建 480 个定时器。

表 11 task. c 函数列表及其功能

函数名	功能	备注
initTaskList()	初始化任务列表	参数略，下同
taskAlloc()	任务项分配	

taskRun()	运行任务

续表 49

函数名	功能	备注
taskSwitchLimit()	手动切换任务	
taskSwitch()	自动切换任务	
taskSwitchSub()	任务切换子程序	
taskSleep()	任务休眠	
addTask()	增加任务	
removeTask()	移除任务	
idleTask()	空闲任务	
taskNow()	当前任务	

表 12 window. c 函数列表及其功能

函数名	功能	备注
initWindowList()	初始化窗口列表	参数略，下同
windowAlloc()	窗口项分配	
setWindowBox()	窗口设定	
setWindowCaption()	窗口标题设定	
setWindowFocus()	窗口焦点设定	
getWindow()	获取窗口	

表 13 console. c 函数列表及其功能

函数名	功能	备注
consoleTask()	控制台主进程	参数略，下同
newCMDLine()	控制台换行	
sysprint()	单个字符打印	
commandCMD()	控制台命令处理器	
clsCMD()	控制台清屏命令	
dirCMD()	文件列表命令	
typeCMD()	文件查看命令	
sysinfoCMD()	系统信息查询命令	

versionCMD()	系统版本查询命令
memCMD()	内存信息查询命令

续表 51

函数名	功能	备注
runCMD()	运行可执行文件命令	
invalidCMD()	无效命令处理器	
printLine()	打印固定长度分割线	
printLinex()	打印指定长度分割线	
printAddress()	打印指定内存地址	
sysprintl()	打印一行字符串	
sysprintx()	打印指定长度字符串	
sys_api()	系统 API 接口	

表 14 file. c 函数列表及其功能

函数名	功能	备注
readFAT()	FAT 解码器	参数略，下同
loadFile()	文件装载器	
searchFile()	文件搜索引擎	

表 15 service. c 函数列表及其功能

函数名	功能	备注
tssBMain()	TSS 服务测试器	参数略

表 16 string. c 函数列表及其功能

函数名	功能	备注
convertToUppercase()	转为大写字母	参数略，下同
convertToLowercase()	转为小写字母	

表 17 bootpack. c 函数列表及其功能

函数名	功能	备注
NNOSMain()	系统入口函数	无参

附录Ⅶ 系统 API 简表

表 1 系统 API 简表

编号	名称	功能	调用	备注
1	api_printc	打印单个字符	syspirnt()	参数略，下同
2	api_printl	打印一行字符串	syspirntl()	
3	api_printx	打印任意长度字符串	sysprintx()	
4	api_return	返回操作系统	\	
5	api_window	创建窗口	boxDraw8()	
6	api_ascwin	在窗口输出字符	wordsDraw8()	
7	api_boxwin	在窗口绘制色块	boxDraw8()	
8	api_initmalloc	初始化应用程序内存	initMemery()	
9	api_malloc	为应用程序申请内存空间	memeryAlloc4k()	
10	api_free	释放应用程序空间	memeryFree4k()	
11	api_point	在应用程序中绘制像素点	pointDraw8()	
12	api_refreshwin	刷新应用程序窗口图层	refreshCover()	
13	api_linewin	在应用程序窗口绘制彩线	\	
14	api_closewin	释放应用程序窗口图层	freeCover()	
15	api_getkey	获取用户键盘输入	\	
16	api_alloctimer	为应用程序申请定时器	timerAlloc()	
17	api_inittimer	初始化应用程序定时器	initTimer()	
18	api_settimer	设定应用程序定时器	setTimer()	
19	api_freetimer	释放应用程序定时器	freeTimer();	
123456789	\	越权，测试不可能的 API 漏洞	\	测试完成废弃

[注]本表将随系统迭代增减。

附录Ⅷ 系统升级日志

表 1 系统升级日志

版本	主要更新（简略版）	日期
NNOS_0.00a	十六进制编辑器编写的 img 镜像，在裸机屏幕显示少量字符信息。	2019.3.2
NNOS_0.01a	汇编语言重写部分代码，生成 img 镜像，在裸机屏幕显示少量字符串信息。	2019.3.2
NNOS_0.01b	汇编语言重写全部代码，生成 img，在裸机屏幕上显示大量字符串信息。	2019.3.3
NNOS_0.02a	新增简易引导程序，生成镜像，在裸机屏幕显示信息。	2019.3.3
NNOS_0.03a	完善简易引导程序。	2019.3.4
NNOS_0.04a	新增源文件批处理程序，支持简单的源文件自动化编译和运行。	2019.3.4
NNOS_0.05a	引入 Makefile 文件生成规则，支持智能的源文件自动编译运行、源文件管理。	2019.3.4
NNOS_0.06a	修改版本号命名规则，完善 Makefile，兼容 Linux 平台，新增源文件转换批处理。	2019.3.5
NNOS_0.06b	磁盘初始化测试，从启动程序装载器到核心入口，初始化磁盘。	2019.3.5
NNOS_0.06c	初级读盘测试，读到 C0-H0-S18 扇区。	2019.3.5
NNOS_0.06d	中级读盘测试，读到 C0-H0-S18 的下一扇区，即 C0-H1-S1。	2019.3.6
NNOS_0.06e	IPL 完工，进入 OS 开发阶段。引导区读盘，启动 NNOS。重构、优化 Makefile。	2019.3.6
NNOS_0.06f	使用 IPL 引导 NNOS，引导程序添加 NNOS 装载位置的跳转	2019.3.6
NNOS_0.06g	NNOS 实模式支持显示字符画面	2019.3.6
NNOS_0.06h	调用硬件 BIOS 中断，设置显卡模式、获取键盘状态、保存硬件信息到内存。	2019.3.6
NNOS_0.06i	进入 C 语言开发阶段，转入保护模式开发阶段。升级 Makefile，新增 getnas。	2019.3.7
NNOS_0.06j	修改 Makefile，兼容 Linux。新增 osfun。	2019.3.8
NNOS_0.07a	升级 osfun，增加“write_mem8()”，支持内存写入。支持显示器白屏。	2019.3.9
NNOS_0.07c	支持显示黑白花纹，使用 C 语言指针取代“write_meme8()”。	2019.3.9
NNOS_0.07e	支持彩色花纹屏，测试不同指针形式。	2019.3.9
NNOS_0.07f	升级颜色模式，新增 I/O 中断屏蔽、恢复，8 位、16 位、32 位内存读写函	2019.3.9

	数等。	
NNOS_0.07g	支持绘制矩形图案。新增桌面、任务栏，停机函数独立、画板增强、常量合并。	2019.3.9
NNOS_0.08c	部分函数重构、完善结构体、读取系统信息、修复漏洞 KB00000000。	2019.3.10
NNOS_0.08d	支持绘制、显示字符，IPL 重构、字体库外置。	2019.3.10
NNOS_0.08f	扩充字体库、支持基本 ASCII 字符、移除 FONT 结构体，重写字体渲染引擎。	2019.3.11
NNOS_0.08i	GDT 初始、IDT 初始化。	2019.3.12
NNOS_0.09c	优化 Makefile，重构源代码，新增中断处理程序，废除多个结构体并入 BOOT_INFO。	2019.3.12

续表 57

版本	主要更新（简略版）	日期
NNOS_0.09d	初始化 PIC,支持简单的鼠标指针移动、新增 nnos.h，废除 osfun.h 并入 nnos.h。	2019.3.12
NNOS_0.09e	处理键盘、鼠标中断，升级 Makefile。	2019.3.12
NNOS_0.10a	支持获取按键编码，中断处理优化，新增 FIFO 缓冲区。	2019.3.13
NNOS_0.10e	继续处理键盘、鼠标中断，支持接收鼠标数据、修复漏洞 KB00000001。	2019.3.13
NNOS_0.10g	处理鼠标中断，初始化 KCMD，键盘控制器切鼠标模式，获取键盘、鼠标中断数据。	2019.3.14
NNOS_0.11c	解读鼠标中断数据	2019.3.14
NNOS_0.11d	支持鼠标移动，优化鼠标指针移动算法。基本 I/O 管理系统完成。	2019.3.15
NNOS_0.12a	重置 SCRENN，重构并分割源文件、独立设备信息文件、升级 getnas 和 Makefile。	2019.3.15
NNOS_0.12d	支持内存容量检查，新增内存管理系统	2019.3.15
NNOS_0.13b	新增 memery.c，支持内存 4K 对齐，鼠标窗口叠加处理，废弃 SCREEN 且永不启用。	2019.3.20
NNOS_0.14a	提高图层叠加处理速度，只刷新变化区域。	2019.3.21
NNOS_0.14d	修复屏幕右边缘鼠标显示、支持任意窗口显示、新增窗口常量、新增窗口创建引擎。	2019.3.21

NNOS_0.15d	支持 PIT，新增定时器，新增 timer.c，重构 BOX，重写 BOX 相关所有源代码	2019.3.21
NNOS_0.16b	简化字符串绘制，清理源文件注释，精简缓冲区，精简定时器，支持标签绘制。	2019.3.22
NNOS_0.16f	系统定时器性能测试对比，生成测试报告	2019.3.22
NNOS_0.16g	合并键盘鼠标 PIT 缓冲区，规定中断信号范围，新增通用 FIFO。修复 KB00000002。	2019.3.23
NNOS_0.16h	优化中断处理速度，优化定时器，清理注释，改进测试方法重写测试定时器	2019.3.23
NNOS_0.16i	精简定时器中断控制算法，精简定时器设定算法，修复 FIFO 溢出漏洞 KB00000003。	2019.3.24
NNOS_0.17e	性能测试，支持高分辨率，清理注释，修复 syshead 兼容问题，升级 getnas。	2019.3.24
NNOS_0.17h	支持键盘输入，支持简单窗口移动。	2019.3.24
NNOS_0.17i	支持追加输入，支持简单文本编辑。	2019.3.25
NNOS_0.18d	支持简单多任务，支持多任务窗口显示，多任务性能测试。	2019.3.30
NNOS_0.18e	加快多任务处理速度，多任务性能测试。	2019.3.31
NNOS_0.18f	分离多任务模块，新增 task.c。多任务自动管理，新增进程管理系统。	2019.3.31
NNOS_0.19a	清理注释，进一步优化进程管理。	2019.3.31
NNOS_0.19b	支持任务休眠，更改通用 FIFO 结构体，新增休眠唤醒标志位。	2019.3.31
NNOS_0.19c	支持统一程序多个实例，清理注释，支持 U 盘启动，支持物理机运行。	2019.4.1
NNOS_0.19e	新增优先级队列。	2019.4.2

续表 57

版本	主要更新（简略版）	日期
NNOS_0.20a	新增控制台，清理注释。	2019.4.2
NNOS_0.20d	控制台支持字符输入，支持多窗口切换和基本多窗口字符输入。	2019.4.3
NNOS_0.20f	完善键盘字符映射表，引入 996ICU 协议，支持大小写切换，支持 LED 等状态控制。	2019.4.3
NNOS_0.21c	改进光标闪烁，响应回车键，注释清理。	2019.4.3

NNOS_0.21d	控制台支持滚动显示。	2019.4.3
NNOS_0.21f	控制台支持命令执行，新增“mem”命令和“clear”命令。	2019.4.3
NNOS_0.21g	显示磁盘数据，新增多个命令。修复控制台显存污染，修复无效字符引用省略。	2019.4.4
NNOS_0.22a	新增“type”命令，新增基本的文件管理系统。解决“type”无拓展名无效问题。	2019.4.4
NNOS_0.22b	处理特殊字符，清理注释。	2019.4.4
NNOS_0.22c	支持 FAT，调整文件管理系统使适应 FAT。	2019.4.5
NNOS_0.22e	分割源文件，新增 console.c、file.c、service.c 等，支持.nex 应用程序。	2019.4.5
NNOS_0.23d	完善进程管理，新增 API，修复部分命令异常。解决多个控制台问题。	2019.4.6
NNOS_0.23h	通过 IRQ 提供 API，使用 sysprintl 代替部分 lableDraw。规定 API 功能号。	2019.4.6
NNOS_0.24a	解决字符显示 API 问题，清理源文件注释。	2019.4.6
NNOS_0.24c	支持 C 语音编写的应用程序。	2019.4.6
NNOS_0.24d	保护操作系统，防止应用程序执行越权指令。应用程序段与系统程序段分开。	2019.4.6
NNOS_0.24e	继续完善操作系统保护，新增一般异常中断处理器。	2019.4.9
NNOS_0.24g	屏蔽 DS 寄存器攻击。	2019.4.9
NNOS_0.25a	系统攻防测试。	2019.4.9
NNOS_0.25e	新增显示字符的 API，修改.nex 编译程序，重置标识。	2019.4.10
NNOS_0.25g	支持应用程序可视化，新增应用程序内存申请 API，窗口应用实例测试。	2019.4.11

[注]本表将随系统迭代不断增加。

参考文献

[1] 川合秀实.30 天自制操作系统 [M].北京:人民邮电出版社,2012.

[2] 于渊. Orange S:一个操作系统的实现[M].北京:电子工业出版社,2009.

[3] Intel 官方.Intel80x86 处理器开发者手册[H].百度网盘,2001

[4] 汤子瀛. 计算机操作系统 第四版[M].西安电子科技大学出版社,2014.

[5] 李忠,王晓波,余洁.x86 汇编语言：从实模式到保护模式[M].北京:电子工业出版社,2012.

[6] 赵炯.Linux 内核完全注释[M].北京:机械工业出版社,2004.

[7] Linus Torvalds. Linux2.6 内核源码[CP].git.kernel.org,2003.

[8] Belady, L. A. .An anomaly in space-time characteristics of certain programs running in a paging machine [J]. Communications of the ACM, 1969.

[9] 水头一寿.CPU 自制入门[M].北京:人民邮电出版社,2014.

[10] Alfred V.Aho,Monica S.Lam 等.编译原理(第二版)[M].北京:机械工业出版社,2009.

[11] 兰德尔 E.布莱恩特.深入理解计算机系统(第三版)[M].北京:机械工业出版社,2016.

[12] James Molloy. UNIX-clone OS[CP].www.jamesmolloy.co.uk,2008.