西南林业大学

本科毕业（设计）论文

（二〇一八届）

题　　目： 操作系统探索

分院系部： 大数据与智能工程学院

专　　业： 计算机科学与技术专业

姓　　名： 尹志成

导师姓名： 王晓林

导师职称： 讲师

二〇一八年六月

操作系统探索

尹志成

（西南林业大学 大数据与智能工程学院, 云南昆明 650224）

摘　要： 操作系统最初的诞生是为了搭配进行简单繁重的数字运算机, 但随着时代的演进, 计算机不仅作为处理各种运算的机器, 其附加价值也越来越被人们看重, 随着计算机的发展, 多道程序系统, 分时操作系统, 一个个新技术的诞生标志着操作系统越发的完善, 但是为了探究操作系统, 那就不应该满足现有的操作系统, 应该拨开操作系统华丽的外衣, 从底层开始, 经过一步步的摸索, 不断探索并实现各项功能, 从最简洁的代码完成一个满足现代各项需求的操作系统。

关键词： 操作系统 设计 原理 简化

Operating system exploration

Zach Yin

School of Big Data and Intelligence Engineering
Southwest Forestry University
Kunming 650224, Yunnan, China

Abstract: The initial version of the operating system was to collaborate with simple and heavy-duty digital computing machines. However, with the evolution of the times, computers have not only been used as machines for processing various operations, but also added more value with people. With the development of computers, The multi-programming system, the time-sharing operating system, and the emergence of new technologies mark the increasingly perfect operating system. However, to explore the operating system, it should not satisfy the existing operating system. It should open up the gorgeous operating system. Starting from the ground floor, after a step-by-step exploration, we constantly explored and implemented various functions, completing an operating system that satisfies modern requirements from the most concise code.

Key words: OperatingSystem Design Principle simplify

目　录

插图目录

表格目录

1 引言

在数字时代, 操作系统的重要性不言而喻, 它作为计算机软硬件之间的桥梁, 存在于日常生活的每一个角落, 人们的衣食住行都无法与各种各样的系统完全脱离。

操作系统在各种应用场合向各种应用程序提供运行环境:

办公 无纸化办公已经成为现代企业必备, 各种文档都保存到公司的电脑和服务器中, 这些终端保存文档的过程离不开操作系统; 企业办公必备的 Email, 打卡等也都必须依赖于操作系统;

移动 看新闻离不开新闻 APP, 点外卖离不开外卖 APP, 连看时间和设置闹铃的时钟也都需要运行在操作系统环境上;

出行 出行方式包括车辆, 船只, 飞机等, 而现代调度平台已经全部电子化, 即调度工作依赖于操作系统。

可见, 操作系统在现代社会的必要性, 而作为一个被依赖如此严重的软件, 其设计的复杂程序是常人难以想象的, 通常只有庞大的互联网科技公司聚集成百上千的高级工程师花费无法计数的日月才能完成一个操作系统, 而设计并完成一个操作系统对于一个学生而言是一个几乎不可能完成的挑战, 但是想要提高理论与技术能力就必须克服难关, 勇敢的跨出这一步, 所以设计并完成一个基本满足日常功能需求的操作系统作为此次的目标, 并以此为跳板对操作系统进行更深一步的探究。

2 技术路线

此次的技术路线由 3 步逐步进行:

1. 操作系统探究
2. 编写操作系统内核
3. 对外兼容及安全防护

2.1 操作系统探究

操作系统的发展是跟随计算机性能的提升一步步趋于完善的, 从历史上第一台计算机的诞生直到今天的计算机, 计算机的性能从处理字节级别到胜任各种高级任务, 操作系统的功能也从手动调度计算机硬件到了操作系统自动调度各种高级程序使用相关硬件, 并对用户需求的各项功能提供接口。

随着计算机走入寻常百姓家, 计算机需要面对的生活需求越来越多, 计算机操作系统随之发展以适应各种工作环境的各种工作需求, 这样的发展态势使得操作系统的功能越来越多, 各项功能也逐渐趋于完善, 但是操作系统出现问题的概率也随着代码数量的增加而增加, 联系到人们的需求, 寻求符合操作系统发展且适应用户使用的功能, 并将其他功能作为可选项排除在操作系统内核之外已经越来越成为趋势。

2.2 编写操作系统内核

从探索的结果开始设计并实现功能操作系统功能。

空白操作系统启动: 计算机用户平时启动计算机操作是点按电源键, 然后等待计算机开机并出现习惯的操作系统界面出现, 并开始借助计算机完成各项工作。对普通用户而言按了开机键后操作系统启动是理所应当的, 但是其中的过程普通用户看不见, 是完全封闭的黑匣子。打开这个黑匣子, 利用操作系统相关知识并查阅相关资料探究操作系统从电气设备到软件代码的衔接则是探究操作系统的第一步。

内存管理：内存是现代计算机的最基础的部件之一, 程序的运行必须加载到内存中, 所以在完成操作系统启动的下一步就是处理好内存的管理问题。内存的管理涉及到内存空间的初始化, 分配和释放问题, 首当其冲的是对内存空间的初始化, 即按照一定的大小将内存分割成若干份。然后对申请内存的程序按照其需求大小分配内存空间, 在程序运行结束后释放其内存空间。

好的内存管理方式对计算机运行效率至关重要:

1. 如果初始化内存指定大小过大将导致多程序运行时每个程序占用内存过剩, 无空闲内存给新程序; 初始化大小过小则导致内存管理程序任务过重, 需要花费过多的资源用于管理内存;
2. 如果分配过程中将内存分配过于分散也将导致管理内存程序花费过多资源;
3. 在释放内存空间的过程中, 算法设计不合理将导致磁盘空间碎片化严重, 而如果算法错误导致程序出错指针等指向错误甚至可能导致系统崩溃。

输入输出：计算机的出现是为了处理一些用户不想处理甚至无法处理的任务。所以计算机与用户的交互在计算机运行过程中也很重要, 计算机的运行任务是用户给予的, 计算机的运行结果也是为了给用户的。用户与计算机进行数据交流的最基础方式是键盘输入, 而后也出现了鼠标的图形化输入。键盘输入的精准性一直被世人所称赞, 而鼠标的空间方式也为图形化界面用户提供了方便。

多进程及分时系统：初代计算机用户只需要运行一个需要花费大量 CPU 运行时间的数字运算, 而现代计算机用户对计算机需求却离不开多任务。

多任务常出现在生活中, 一个人工作时会抽出时间帮他人一个小忙又继续自己的工作, 但是计算机没有人类复杂的大脑, 计算机同时只能完成一个任务。

所以在现代计算机操作系统发展中里程碑式出现了多道程序设计和分时操作系统, 在相当程度上完成了本不可能出现了“同时运行多个程序”, 使多个程序在人类不能明显感觉到的时间间隔内来回切换, 并在切换过程中保存程序运行现场保证下一次程序可以继续运行。

现代计算机多任务的基础技术, 对计算机操作系统的发展至关重要。

2.3 实现对外接口及安全防护

计算机操作系统的功能按常理来说是可以无限扩展的, 但是扩展将导致操作系统无限臃肿, 而后果是操作系统的维护无限困难, 更难以接受的是臃肿的系统将出现无数漏洞暴露给恶意程序。

所以在不削减操作系统功能的基础上, 将操作系统分为内核和外部应用程序, 操作系统内核为外部应用程序提供接口, 以指定的方式让外部应用程序调用系统函数间接完成对硬件的操作, 在系统易于维护且保证安全的基础上为用户提供需要的功能。

系统的安全除了对系统接口的授权外, 也涉及到监控无法正常运行的程序并使其终止, 对损坏硬件的检测等方面。

3　操作系统探索

3.1　操作系统的诞生

3.1.1　第一代:真空管

操作系统最初出现的场景是一个工程师小组设计、建造一台机器, 之后使用机器语言编写程序并通过将上千根电缆接到插线板上连接成电路, 控制机器的基本功能, 进而操作机器运算诸如制作正弦、余弦、对数表或计算炮弹弹道的简单数学运算。

这里的人工拔插电缆就充当着操作系统的角色——根据程序直接操作硬件使其运算得出结果。

3.1.2　第二代:晶体管

在晶体管发明后, 计算机可靠程度大大增加, 计算机开始被一些公司、政府部门或大学使用。

改进后出现了操作系统的载体, 卡片和较后期磁带打孔纸带。

由于打孔纸带是分次读入, 一次只能读入一个的作业, 出现了批处理系统如图 3-1:

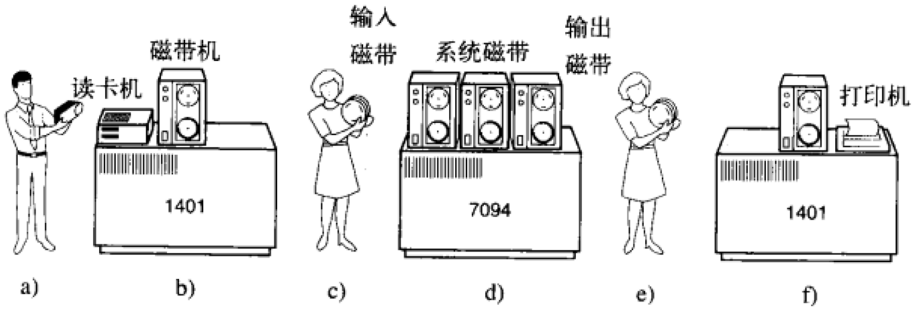

图 3-1　批处理系统

a. 程序员将打孔纸带拿到 1401 机[1]

b. 1401 机将批处理作业读到磁带上

[1]IBM 1401:数据处理计算机[1]

c. 操作员将输入磁带送至 7094 机[2]

d. 7094 机进行计算

e. 操作员将输出磁带送到 1401 机

f. 1401 机打印输出

在这里, 操作系统的工作已经由完全的人工转换到一部分人工操作交由机器完成, 工作效率较之前大大提高, 并且, 由于加入了磁带, 计算机完成的工作也将及时得到保存。

操作系统的工作已经开始有一定的流程化了。

3.1.3 第三代:集成电路

采用集成电路的第三代计算机较分立晶体管的第二代计算机在性能/价格比上有了很大的提高,

第三代操作系统加入了多道程序设计和分时系统, 可以适应多道程序同时运行的任务。

1. 多道程序设计主要目的是解决 CPU 因等待磁带或其他 I/O 操作而暂停工作, 多道程序设计可以使 CPU 在程序 a 的 I/O 操作时运行程序 b[3]。
2. 分时系统解决的主要问题是多用户使用分离的终端, 却操作同一台计算机。

3.1.4 第四代:个人计算机

大规模集成电路进一步减小了计算机的大小, 进一步为计算机进入大众视野做好了铺垫。

现在使用的台式机和笔记本就是属于第四代计算机发展的较高版本。

第四代计算机和之前的计算机比较而言在于同一台终端的使用人数, 第四代之前的计算机主要用于科研和大规模计算, 所有硬件都是为这些作业而工作, 操作系统也只用为这些作业安排调度, 而每个人都能拥有自己的计算机, 那么计算机就需要有更多个性化的东西, 这对操作系统提出了更多的要求:

1. 更多的进程
2. 更大的存储空间(文件管理)

[2]IBM 7094: 专为大型科学计算而设计, 具有出色的性价比和扩展的计算能力[2]

3. 更多设备接入(如鼠标)
4. 网络要求的实现
5. 对外防护的要求
6. 对外提供接口(图形化显示)

3.1.5 第五代:移动计算机

第五代计算机是便携式计算机, 也就是智能手机或者智能平板设备, 较之第四代计算机最大的不同在于易于携带, 但是缺点也很显著, 受到设备体积和电池的限制, 性能受到很大的影响。

第五代计算机对操作系统有了新的要求:

1. 从待机状态极快的响应
2. 电量控制(即低成本完成作业)

3.1.6 小结

纵观操作系统的发展史, 发展的中心问题是“如何更低成本完成更多的任务”, 而发展的关键节点是多道程序设计以及分时系统。

展望计算机的发展史,大胆的推测下一代计算机可能就是”computer everywhwere”, 而那时操作系统也会有新的发展。

3.2 操作系统的规范化

3.2.1 Unix 及类 Unix

Unix 及类 Unix 将用户空间与系统空间划分开, 以此规定内核的边界, 将存在于系统空间的代码与数据的集合称为内核。由此也存在了不同的 CPU 运行模式:系统态和用户态。

1. 系统调用接口
2. 进程管理
3. 内存管理
4. 虚拟文件系统

5. 网络堆栈
6. 设备驱动程序

3.2.2 Windows

Microsoft Windows 与 Unix 最大的不同是, 它将较低层的离硬件最近的一部分叫做“内核”, 因此, Windows 也将图形界面和视窗机制的实现也放在了内核中。

大体来说, Windows 内核层次如下:

1. 系统调用接口
2. 中断/异常入口
3. Executive (管理层)、对象管理、内存管理、进程管理、安全管理、I/O 管理等[4]
4. 核心层、设备驱动底层
5. HAL(硬件抽象层)

3.2.3 小结

从市场中用户数量较多的两大操作系统平台看, 操作系统已经趋于规范化, 功能方面也趋于成熟, 两大操作系统的功能都非常完备, 但功能多了的坏处就是 bug 出现的概率也随之变高, 所以 windows 的蓝屏和 linux 的系统抖动等操作系统问题都对用户造成了极大的困扰。

想要解决这些问题较好的办法就是精简功能, 从降低代码数量开始降低系统 bug 的出现概率, 精简功能则要很好的处理好必要功能和非必要功能的筛选分类。

4　编写操作系统内核

衔接计算机硬件到软件代码的流程, 并扩展到内存管理, 输入输出, 多进程, 分时四个模块丰富操作系统的内容。

4.1　启动空白操作系统

4.1.1　操作系统启动流程

按下电源键后计算机开始启动, 启动过程分为 3 个阶段[5]:

BIOS -> MBR -> 操作系统

1. 在 BIOS 完成 POST（硬件自检, Power-On Self Test）并根据启动顺序（Boot Sequence）来选择启动设备。本系统是从 U 盘启动。
2. 计算机读取该设备的 MBR（Master boot record, 位于第一个扇区, 即最前面的 512 个字节, 见图 4-1）, 并运行其中的启动程序 IPL（Initial Program Loader）, 将 ZOS 加载入内存。本部分的实现代码, 参见附录程序 C.1;
3. 控制权转交给操作系统后, Kernel 开始运行, 操作系统启动完成。

4.1.2　制作 MBR

MBR 的主要部分是 Bootstrap code area, 即前 446 个字节。MBR 结构见图 4-1。负责指出操作系统的位置, 主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）, 参见附录程序C-1。

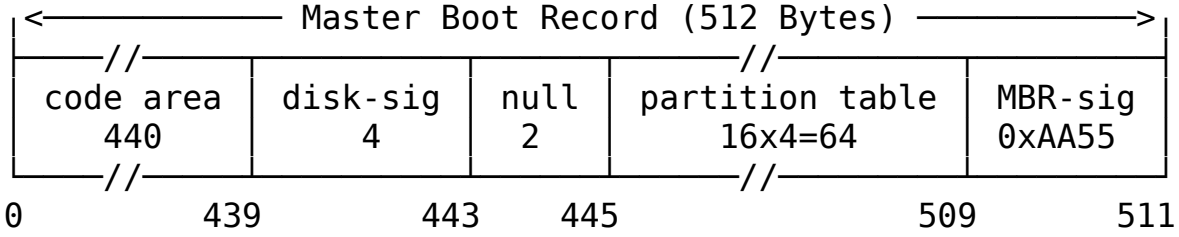

图 4-1　MBR

一个扇区大小为 512 字节, MBR 位于 C0-H0-S1(柱面 0, 磁头 0, 扇区 1),[6] 从下一个扇区 (扇区 2, C0-H0-S2) 开始加载操作系统。找到扇区 2 的操作如程序 **??**所示。

```
43  MOV    CH,0    ; 柱面 0
44  MOV    DH,0    ; 磁头 0
45  MOV    CL,2    ; 扇区 2
```

程序 4-1 初始化读取柱面、磁头和扇区的起点

完成定位后开始将磁盘数据读入内存, 策略是

1. 磁头 0, 柱面 0, 读取 1-18 扇区 (C0-H0-S2)-(C0,H0,S18)
2. 磁头 1, 柱面 0, 读取 1-18 扇区 (C0-H1-S1)-(C0-H1-S18)
3. 磁头 0, 柱面 1, 读取 1-18 扇区 (C1-H0-S1)-(C1-H0-S18)
4. ...
5. 磁头 1, 柱面 78, 读取 1-18 扇区 (C78-H1-S1)-(C78-H1-S18)
6. 磁头 0, 柱面 79, 读取 1-18 扇区 (C79-H0-S1)-(C79-H0-S18)
7. 磁头 1, 柱面 79, 读取 1-18 扇区 (C79-H1-S1)-(C79-H1-S18)

按以上策略将磁盘内容读入内存, 核心代码如程序见 C-3。

readfast 位于代码第 76 行, JMP readfast 在此代表循环执行。当循环执行完毕, 表示磁盘内容已经全部加载到内存中, MBR 过程成功, 开始启动操作系统。

4.1.3 制作空白操作系统

为测试操作系统是否成功被 MBR 启动, 设计将操作系统设置为启动后待机。如程序 4-2所示。按下电源键, 经过启动步骤系统循环执行 HLT[1] 使得操作系统计算机始终处于待机状态, 启动成功。

```
1  fin:
2  HLT
3  JMP fin
```

程序 4-2 空白操作系统

[1]HLT: 让 CPU 停止动作并进入待机状态[7]

4.2 操作系统组件功能全览

在完成空白操作系统后, 操作系统的设计才正式拉开序幕, 此次为操作系统设计的功能如图 4-2所示, 操作系统启动后将完成各种功能需求的初始化操作, 在初始化成功后各项功能就可以正常运行。

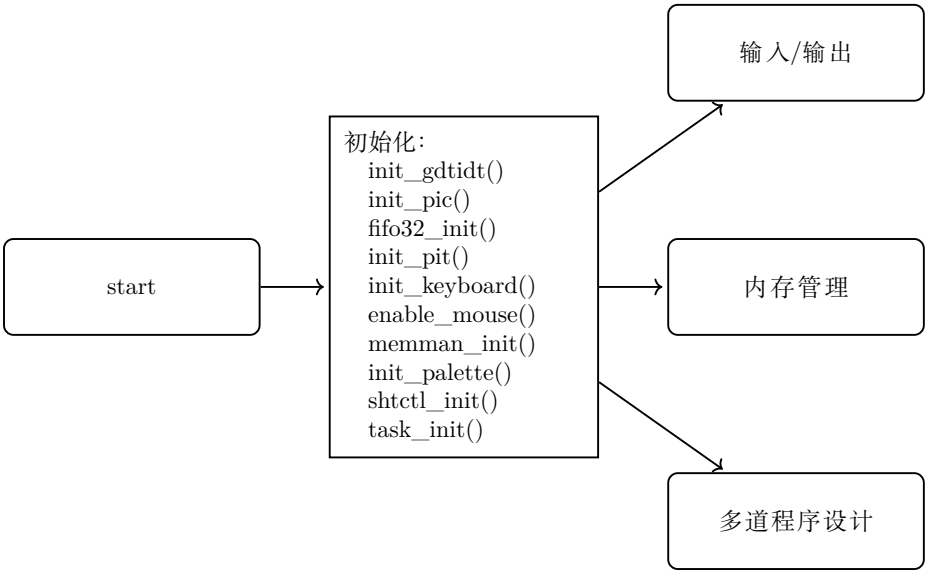

图 4-2 操作系统组件功能全览

4.3 内存管理

内存 (RAM) 是计算机中不可缺少的重要硬件, 所有程序的运行都是在内存中进行的, 而 CPU 访问硬盘数据也必须先经过内存交换才得以实现, 内存在加速 CPU 访问硬盘居功至伟。由内存的重要性可知内存管理在操作系统中也非常重要。

根据内存的特殊性, 内存错误将导致程序或操作系统错误, 故操作系统将在内存管理中首先检测内存是否故障, 确认内存完好后开始对内存进行初始化并清空内存空间内容, 在程序申请内存后向其分配内存, 并程序运行结束后释放其内存。功能设计如图 4-10。

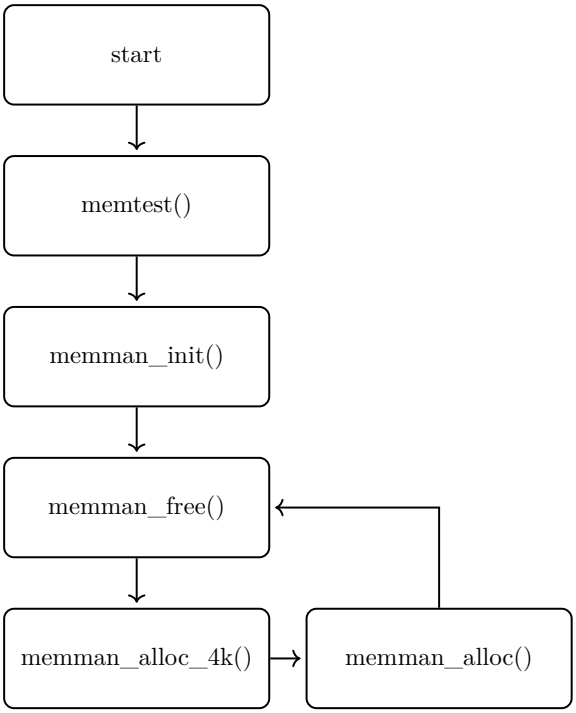

图 4-3 内存管理

内存的结构通常用地址和空间表示, 在计算机运行过程中, 内存的分配是随机的, 导致内存的释放也是相对无序的, 这样导致了很多的碎片化问题, 也就是随计算机运行时间变长, 内存中到处遍布小块零散空闲空间, 虽然零散空间总数很大, 但很难满足新进程序的内存需求, 于是内存管理显得十分必要。

内存管理设计的主要目的是快速并且高效的分配内存空间, 并在适当的时间释放并回收内存空间。根据内存管理的设计目的, 数据结构设计如程序 4-3。

```
137 struct FREEINFO {  /* 剩余容量信息 */
138   unsigned int addr, size;
139 };
140 struct MEMMAN {    /* 内存管理 */
141   int frees, maxfrees, lostsize, losts;
142   struct FREEINFO free[MEMMAN_FREES];
143 };
```

程序 4-3 数据结构-内存管理

frees: 当前可用内存组数

maxfrees: 可用内存组数的最大值

lostsize: 释放失败的内存的大小总和

losts: 释放失败次数

经过内存初始化和释放所有内存空间后, 内存管理正常运行。

4.3.1　内存分配

内存的分配方式涉及到内存释放, 好的分配方式会使得内存使用的效率大大提高。根据内存的大小来划分内存如何使用, 预计使用 32KB 用于内存分配的管理空间, 则共有 4000 组左右的内存用于分配给各个程序使用, 每个组 4KB。

每一组内存经过初始化都拥有自己的数据结构, 即每一组空闲内存的地址和大小都被记录到空闲内存表4-1。

组号	地址
2	0x00005000
1	0x00004000
0	0x00003000

表 4-1　空闲内存表 free

一旦系统接收到程序申请内存的请求(需求的内存大小), 就开始在内存中寻找足够大的内存完成这次申请, 并返回可供使用的空闲内存的地址。完成申请后系统需要重新整理空闲内存表 free, 将可用内存组数减一, 将返回给程序空闲空间大小根据程序需求进行调整, 并对剩余的可用内存表进行按地址升序整理。流程图见4-10, 程序见 C-4。

4.3.2　内存释放

为保证磁盘空闲空间尽可能少的碎片化, 内存释放首先考虑的是使待释放空间与附近空闲空间进行合并[8]。

具体分为三种情况:

前端空闲: 释放内存的相连前端是空闲内存或释放内存相连两端都是空闲内存

后端可用: 释放内存的相连后端是空闲空间

前端后端均不可用: 在当前位置释放内存

已知:待释放的空间的地址和空间大小

根据空闲内存表 free 的编号从 0 到 frees 遍历查找地址大于待释放空间的空闲内存, 并根据得到的空闲内存编号 i 及大小 size 区分此时的待释放内存应当采取何种方式释放, 实现参见程序 C-5。

前端空闲:

当相连前端有可用内存时将可释放内存大小归入前端可用内存内, frees 不变;

当相连后端也有可用内存时将后端内存大小归入前端可用内存内, frees 减一。

内存释放前后情况如图 4-4和图 4-5所示, 实现见程序 C-6:

图 4-4　前端空闲

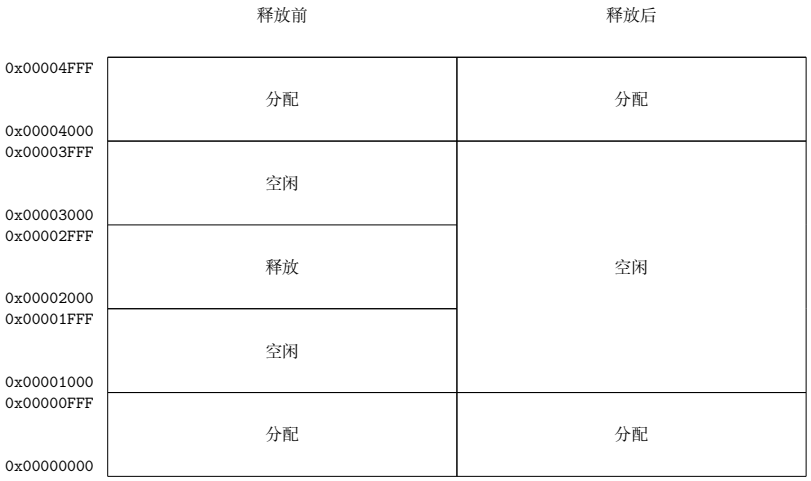

图 4-5　前端可用, 且后端空闲

后端空闲:

当相连后端有可用内存的时候将 free[i] 的地址换为待释放内存的地址, 相连后端内存大小归入待释放内存大小, frees 不变。

内存释放前后情况如图 4-6所示, 实现见程序 C-7。

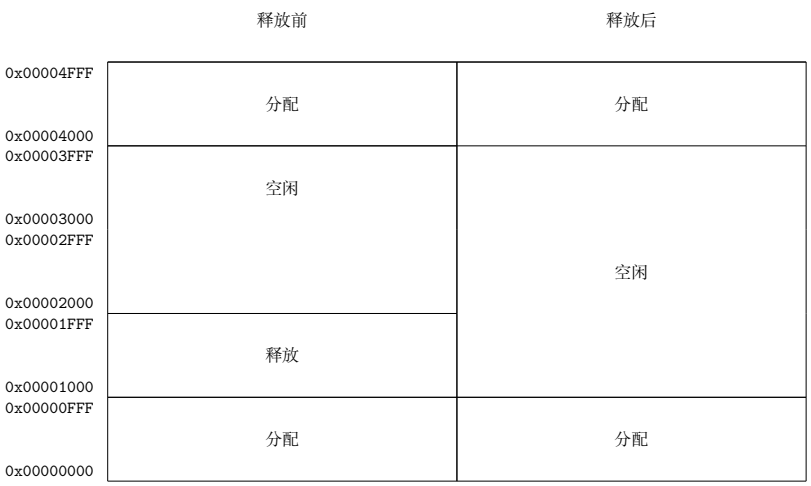

图 4-6　后端空闲

前端后端均被占用:

由于被释放空间周围没有空闲内存, 为保证 free 内各段内存仍然按照内存地址升序排列, 使空闲空间计数最大值加一, free[i] 后续空闲内存序号加一, 并将释放空间组号定为 i。

内存释放前后情况如图 4-7所示, 实现见程序 C-8。

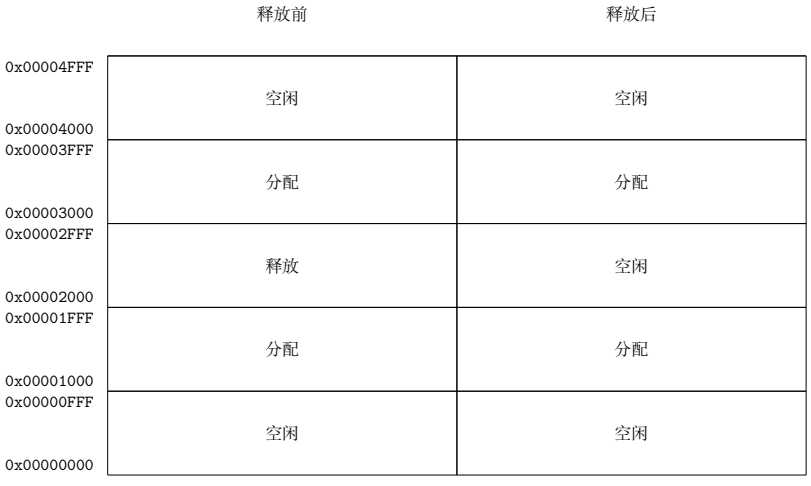

图 4-7　前端后端均被占用

4.4 输入输出

输入作为人与计算机之间最基本的交互方式, 其中键盘和鼠标是标准输入设备。

CPU 通过 PIC[2]（Programmable interrupt controller）与外部设备进行数据交换。键盘作为计算机最早的外设, 位于主 PIC 的 IRQ1; 而鼠标作为第四代计算机才出现的输入设备, 位于从 PIC 的 IRQ12, 而从 PIC 连接在主 PIC 的 IRQ2。

输入设备输入时是将一个个指令[3] 发送到 CPU, 而 CPU 同时只能处理一个指令（虽然处理时间非常短）, 所以需要一个缓冲区 fifo[4]来接收输入数据。缓冲区数据结构见图 4-4, 运行流程见图 4-8。

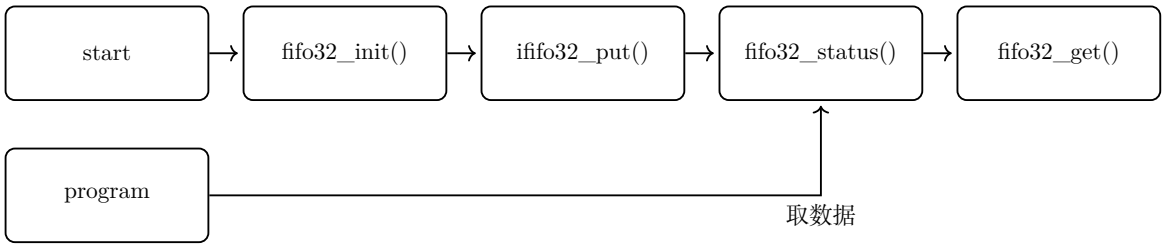

图 4-8 缓冲区

缓冲区在初始化 (fifo32_init()) 成功后开始接受数据 (fifo32_put());

程序调用接收数据需求 (fifo32_get()) 时判断缓冲区 (fifo32_status()) 不为空的情况下取出数据。

```
40 struct FIFO32 {
41   int *buf;
42   int p, q, size, free, flags;
43   struct TASK *task;
44 };
```

程序 4-4 数据结构-缓冲区 fifo

***buf:** 缓冲区在内存中的地址;

[2]PIC 通过中断控制请求使用 CPU, 即在需要的时候对 CPU 发出中断请求, 暂停 CPU 现在运行的作业, 先完成中断请求再继续之前的作业

[3]每一次敲击键盘按键, 移动鼠标, 点击鼠标都会产生响应的指令

[4]缓冲区 fifo 采用先入先出的数据结构, 使得先传入的输入指令能够率先得到执行

p, q: 下一个写入地址, 下一个数据读入地址;

size, free: 缓冲区大小, 空闲空间大小;

TASK flags: 溢出标记(-1 和 0 分别表示有溢出和无溢出);

TASK *task: 在当前位置释放内存。

操作系统运行后始终通过函数 fifo32_status(&fifo)[5] 监测 fifo 缓冲区内是否有数据, 如果有数据才执行下一步对输入指令的响应。接收输入信号的函数 fifo32_get(&fifo) 负责从缓冲区 fifo 中取出一个数据(键盘为 1 个字节, 鼠标为 4 个字节)交由 CPU 处理。流程图见4-8, 程序见C-9.

计算机收到数据后根据数据的大小区间区分这一段数据是何处传来的并进行相应处理。一旦收到的指令为 256 到 511, 系统判定为键盘输入; 当指令为 512 到 767, 系统判定为鼠标输入。

4.4.1 输入输出

输入输出流程如图 4-9所示。

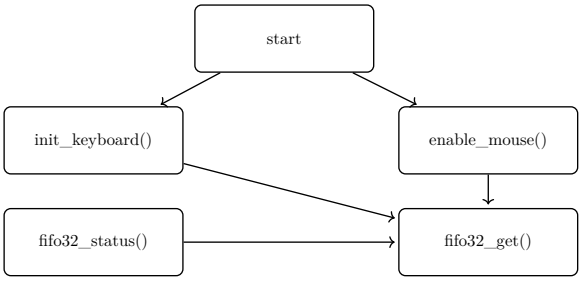

图 4-9 输入输出

操作系统启动后启用键盘输入(init_keyboard())和鼠标输入(enable$_mouse()$) 。

在缓冲区获得输入数据后由程序调用接收数据需求(fifo32$_get()$) 后获得输入数据。

键盘输入

键盘作为最基础也是大众使用最精确的输入设备, 它负担着很多的责任:

1. 输入文本按键
2. 修改文本输入的基本指令按键
3. 修改计算机的状态特殊按键

[5]存储总量 =size-free

按下不同的功能按键, 键盘向计算机发送的指令是不一样的。

若输入指令为 256-0x80+256 时, 为输入文本指令;

输入指令为退格键, 回车键;

修改计算机的状态特殊按键如表 4-2所示。

功能	指令	功能	指令
左 Shift	256+0x2a	CapsLock	256+0x3a
右 Shift	256+0x36	NumLock off	256+0xaa
左 Shift off	256+0xaa	ScrollLock	256+0xaa
右 Shift off	256+0xb6		

表 4-2 修改状态的特殊指令

鼠标输入

鼠标作为第四代计算机才出现的输入设备, 因为涉及到坐标位置的变化, 它的输入指令较键盘也相对复杂, 从鼠标传到计算机的信号都是 4 个字节一组的, 其中第一个字节表示状态, 其他 3 个有效指令分别为 x 坐标, y 坐标以及鼠标按键状态。根据鼠标接受数据的格式, 数据结构设计见程序 4-5。

```
126 struct MOUSE_DEC {
127   unsigned char buf[3], phase;
128   int x, y, btn;
129 };
```

程序 4-5 数据结构-鼠标输入的数据

buf: 存放鼠标一个数据指令的数组;

phase: 当前鼠标工作的阶段;

x, y 缓冲区大小, 空闲空间大小;

btn 按键状态。

4.4.2 标准输出

由于是面对硬件开发操作系统, 无法使用各种成熟的库和函数, 所以输出也只能从修改 VRAM 的一个个像素开始。

流程如下:通过一系列指令后,系统需要向屏幕打印字符,则从数组中读取一个数据,并从字体库(包含基本字体的像素化数组)中找到对应的显示方法,写入 VRAM(Video RAM),在屏幕对应坐标范围内显示数据,循环读取下一个并显示下一个数据。

4.5 多道程序系统与分时操作系统

4.5.1 多道程序系统

现代的计算机已经不仅仅作为数字计算的工具,进入大众生活的计算机被赋予了更多生活上的需求,用户可能在看电影的同时查看电子邮件,也有可能在写论文的时候进入浏览器查询相关资料,但是更重要的是计算机往往在用户不经意间打开防病毒软件等保证用户计算机的安全。

由此可见多进程的工作方式在计算机工作中不可缺少。

但是在实际的处理过程中,计算机并不能同时处理多个程序。

首先要处理的问题是如何运行多个程序,早期的多道程序设计的出发点是充分的利用 CPU,作为输入设备的打孔纸带与 CPU 速度相比差距过大,昂贵的 CPU 常常在等待 I/O 信号而闲置。

具体情形如:A 作业等待磁盘或者其他 I/O 时,CPU 暂停为 A 作业服务,而转向为已完成 I/O 操作的 B 作业服务,等到 B 需要执行下一步 I/O 操作时,CPU 发现 A 作业完成了 I/O 操作,又转向为 A 作业服务,依次循环直到队列中所有作业完成。

由于 A 作业和 B 作业并行存储在计算机内存中,虽然在具体的执行中多次交换先后顺序执行,但两个作业输出在同一个磁盘,故认为是多道程序在计算机中运行。

4.5.2 分时操作系统

分时是使得在用户看来计算机的多道程序同时运行,但是同时运行这也是不可能的。

所以只能采取折中的办法,使得 CPU 在用户不能明显感觉到的时间间隔内切换运行多个程序,在切换后每个程序都能对作业进行一定的处理,在进行多个周期后,各个程序先后完成作业。

“不能明显感觉到的时间间隔内切换”中有两个概念,时间间隔和切换:

1. 时间间隔太长则用户会有明显的卡顿感, 不利于分时概念的实现, 而时间间隔太短则时间不足以让程序响应并完成一定量的工作, 同样不利于分时概念的实现。
2. 切换涉及到保存当前程序的运行状态以便于程序获得 CPU 时间后可以接续上次的任务继续执行。

4.5.3 定时器

定时器的实现是实现分时操作系统的关键, 定时器每隔一段时间就向 CPU 发送中断信号, 并记录发送的次数, 以定时器发送的中断次数作为定时的依据。

管理定时器主要是使用 PIT[6] (Programmable Interval Timer), 之前说到定时器时间太短不利于分时的实现, 故暂定 1 秒中发生 100 次中断。

在实际的运用中, 往往需要用到不止一个定时器, 于是需要对多个定时器进行管理, 数据结构如下:

```
175 #define MAX_TIMER    500
176 struct TIMER {
177   struct TIMER *next;
178   unsigned int timeout;
179   char flags, flags2;
180   struct FIFO32 *fifo;
181   int data;
182 };
183 struct TIMERCTL {
184   unsigned int count, next;
185   struct TIMER *t0;
186   struct TIMER timers0[MAX_TIMER];
187 };
```

程序 4-6 数据结构-多定时器

Timer *next: 下一个定时器地址;

[6]PIT 连接 IRQ0, 设定 PIT 就可以设定 IRQ0 的中断间隔, 中断频率 = 单位时间事钟周期书(主频)/设定的数值

timeout: 下一个超时时刻;

flags, flags2: 各个定时器的状态, 在应用程序结束时定时器是否取消的标记

count, next: 当前时刻, 下一个时刻;

Timer *t0: 所有时刻都要减去这个值。

对每个定时任务设置到超时时刻(对每个任务的运行时间进行计量, 超过指定时刻向系统发送指令), 这样可以对一些设定了定时的任务在时间到了之后执行指定操作, 如光标的闪烁, 页面的刷新等。

4.5.4　任务切换

在多道程序系统和分时操作系统分时操作系统完成后, 任务切换就可以进行了。

任务切换从字面上理解很简单, 多个任务之间来回切换, 但是任务切换在计算机中实现却不那么简单。

如现有 A 和 B 两个任务, A 正在运行, 现在需要切换到 B。

1. 任务 B 向 CPU 发送切换任务的指令
2. CPU 把当前寄存器中的值全部写到内存中
3. CPU 执行任务 B
4. CPU 切换到任务 A 并把所有寄存器中的值从内存中读出来, 继续执行任务 A

流程见图 4-11。

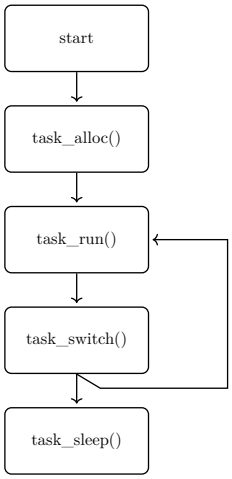

图 4-10　任务切换

为了实现复杂的任务管理, 需要用到的数据结构如下:

```
227 struct TASKCTL {
228   int now_lv; /* 现在活动中的 LEVEL */
229   char lv_change; /* 在下次任务切换时是否需要改变 LEVEL */
230   struct TASKLEVEL level[MAX_TASKLEVELS];
231   struct TASK tasks0[MAX_TASKS];
232 };
```

```
222 struct TASKLEVEL {
223   int running; /* 正在运行的任务数量 */
224   int now; /* 这个变量用来记录当前正在运行的是哪个任务 */
225   struct TASK *tasks[MAX_TASKS_LV];
226 };
```

```
209 struct TASK {
210   int sel, flags;    /* sel 用来存放 GDT 的编号 */
211   int level, priority; /* 优先级 */
212   struct FIFO32 fifo;
213   struct TSS32 tss;
214   struct SEGMENT_DESCRIPTOR ldt[2];
215   struct CONSOLE *cons;
216   int ds_base, cons_stack;
217   struct FILEHANDLE *fhandle;
218   int *fat;
219   char *cmdline;
220   unsigned char langmode, langbyte1;
221 };
```

程序 4-7 数据结构-多任务

4.5.5 任务优先级

在实际的运用中, 所有任务在同一个优先级的安排显然是不合理的, 比如系统在执行 A 作业时, 由于 A 作业和键盘输入作业优先级相同, 所以系统可能认为应该先完

成 A 作业再执行键盘输入。

而通常人为操作中键盘优先级应该是最高的,为此设计任务的优先级。

```
222 struct TASKLEVEL {
223   int running; /* 正在运行的任务数量 */
224   int now; /* 这个变量用来记录当前正在运行的是哪个任务 */
225   struct TASK *tasks[MAX_TASKS_LV];
226 };
```

程序 4-8 数据结构-任务优先级

设计中,将任务分为多个层次,将音乐、网络传输等对优先级要求高的任务放在最高的优先级,将鼠标等任务放在稍低的优先级,将对优先级要求低的任务放在较低的优先级。

则系统在多任务同时运行的情况下会优先处理高优先级的任务,处理完成后处理稍低的优先级任务,优先级较低的任务将在系统较空闲(即无更高优先级任务)的时候处理。

5　对外兼容及安全防护

从接口设计及安全防护的角度完善操作系统

5.1　系统调用

操作系统是用于管理硬件的软件, 但是对绝大部分人都不算友好, 可友好不是操作系统必须的, 操作系统只需要快速和高效, 为此操作系统向外界开放接口, 由其他的编程人员来使用接口完成各种功能的实现, 比如:图形界面, 各种字处理软件等。

系统调用的实现是通过程序向操作系统申请权限访问各种指定的系统函数, 操作系统操作硬件完成工作。

向外界开发的系统调用接口:

1. 显示单个字符
2. 显示字符串
3. 键盘输入
4. 定时器
5. 文件操作
6. 命令行

利用这些接口可以实现程序如下:

1. 命令行计算器(调用键盘输入、显示字符串、命令行)
2. 文本阅读器(调用文件操作)
3. 音乐播放器(调用命令行、文件操作、定时器)
4. 图片阅读器(调用命令行、文件操作)

5.2　系统安全

只有完全封闭的系统才有可能成为真正安全的操作系统, 但是既然选择了对外开放接口, 就必须重视外部应用程序的各种有意或无意的操作给操作系统带来的安全隐患, 一旦有程序跨越操作系统直接操作硬件很可能使得操作系统崩溃。

所以操作系统在开放接口的同时也需要一定的安全举措:

1. 内存写保护: 若将内存直接暴露给应用程序使得应用程序可能不慎在错误的地方写入错误的值, 那么导致的问题可能包括系统崩溃、文件损坏, 甚至导致计算机硬件损坏
2. 系统函数权限: 管理员的权限是通过各个系统函数实现的, 将系统函数开放给对系统不熟悉的用户是完全不负责任的)
3. 监测并终端异常程序: 计算机病毒是自发的运行并对计算机造成意想不到后果的恶意程序, 所以在操作系统发现有程序活动异常时应该立即中断此程序
4. 系统调用防护: 系统调用的目的是使得外部开发者通过系统设计者的意愿得到希望得到的系统函数执行权而开发应用程序, 于是在系统调用方面, 每一个系统调用接口都应该绑定固定的系统函数, 并限制其用法, 拒绝非系统调用的方式启用系统函数。

参考文献

[1] IBM, 1401 Data Processing System, **2003**.

[2] IBM, 7094 Data Processing System, **2007**.

[3] A. S. Tanenbaum, *Modern operating system*, Pearson Education, Inc, **2009**.

[4] 毛德操, Windows 内核情景分析 (上下), **2005**.

[5] 阮一峰, 如何变得有思想, **2014**.

[6] 刘伟, 数据恢复技术深度揭秘, **2010**.

[7] K. Hidemi, *30 天完成! OS 自作入門*, minabi publication, **2006**.

[8] R. E. Bryant, O. David Richard, O. David Richard, *Computer systems: a programmer's perspective*, *Vol. 2*, Prentice Hall Upper Saddle River, **2003**.

指导教师简介

王晓林, 男, 49 岁, 硕士, 讲师, 毕业于英国格林尼治大学, 分布式计算系统专业。现任西南林业大学大数据与智能工程学院教师。执教 Linux、操作系统、网络技术等方面的课程, 有丰富的 Linux 教学和系统管理经验。

致　谢

四年时光匆匆而过, 解了不多不少的惑。一初听说最怕的人, 成了感谢最多的人。

当过班上班主任的王晓林老师, 在我心里仍然是班主任, 我想这不止是因为他当过, 而是我认为他当过。他的认真负责和较真较劲让上过课的每个学生印象深刻, 他教书教得好, 是大家都认同的, 而这里的大家还包括一些讨厌他古板作派的同学。他的一门课只有一个 presentation, 学期结束了, presentation 也就结束了, presentation 的内容不跟教学大纲走, 在大学课本里也都找不到, 但是总能在权威的技术书籍里看到影子, 也总能在相关知识的学习中联想到并且用到。他上课不喜欢教, 就喜欢问, 然后像个学生一样跟学生讨论来讨论去, 讨论着讨论着我们就学会了, 而且印象深刻。他崇尚 GNU 的自由精神, 也不断告诉我们觉得对的就应该不顾一切勇敢的努力的去做。他也喜欢干点专权的事, 拿着老师的架子逼着全班学生装 Linux, 也许之前也想不通为什么要这样做, 但是如果没有这样一个人逼着改变, 我想 Linux 对我只是一门课, 而现在 Linux 是我提高效率的工具。

选择毕业设计导师的时候我没怎么犹豫, 我想, 对, 就是他。

附录 A　文件清单

A.1　根目录

app_make.txt apps/ demos/ fonts/ includes/ kernel/ libs/ media/ ruls/

A.2　Kernel 目录

asmhead.nas bootpack.c bootpack.h console.c dsctbl.c fifo.c file.c graphic.c hankaku.txt int.c ipl09.nas keyboard.c Makefile memory.c mouse.c mtask.c naskfunc.nas sheet.c tek.c timer.c window.c

A.3　includes 目录

apilib.h errno.h float.h golibc.lib harilibc.lib limits.h math.h setjmp.h stdarg.h stddef.h stdio.h string.h

附录B　函数功能

附录C　主要程序代码

C.1　初始启动程序代码(ipl09.nas)节选

```
12  DB    " zbote  "  ; 启动扇区名称(8 字节)
13  DW    512     ; 每个扇区(sector)大小(必须 512 字节)
14  DB    1       ; 簇(cluster)大小(必须为 1 个扇区)
15  DW    1       ; FAT 起始位置(一般为第一个扇区)
16  DB    2       ; FAT 个数(必须为 2)
17  DW    224     ; 根目录大小(一般为 224 项)
18  DW    2880     ; 该磁盘大小(必须为 2880 扇区 1440*1024/512)
19  DB    0xf0     ; 磁盘类型(必须为 0xf0)
20  DW    9       ; FAT 的长度(必须为 9 扇区)
21  DW    18      ; 一个磁道(track)有几个扇区(必须为 18)
22  DW    2       ; 磁头数(必须为 2)
23  DD    0       ; 不使用分区,必须是 0
24  DD    2880     ; 重写一次磁盘大小
25  DB    0,0,0x29  ; 意义不明(固定)
26  DD    0xffffffff  ; (可能是)卷标号码
27  DB    "    ZOS    "  ; 磁盘的名称(必须为 11 字节,不足填空格)
28  DB    "FAT12   "  ; 磁盘格式名称(必须为 8 字节,不足填空格)
29  RESB  18       ; 先空出 18 字节
```

程序 C-1　FAT12 格式磁盘专用代码

```
76  readfast:  ; 使用 AL 尽量一次性读取数据 从此开始
77  ; ES: 读取地址, CH: 柱面, DH: 磁头, CL: 扇区, BX: 读取扇区数
78
79      MOV     AX,ES       ; < 通过 ES 计算 AL 的最大值 >
80      SHL     AX,3        ; 将 AX 除以 32,将结果存入 AH(SHL 是左移位指令)
81      AND     AH,0x7f       ; AH 是 AH 除以 128 所得的余数(512*128=64K)
82      MOV     AL,128        ; AL = 128 - AH; AH 是 AH 除以 128 所得的余数
        ↪ (512*128=64K)
83      SUB     AL,AH
84
85      MOV     AH,BL       ; < 通过 BX 计算 AL 的最大值并存入 AH >
86      CMP     BH,0        ; if (BH != 0) { AH = 18; }
87      JE     .skip1
88      MOV     AH,18
```

程序 C-2　将磁盘内容读入内存

```
138 ADD     CL,AL     ; 将 CL 加上 AL
139 CMP     CL,18     ; 将 CL 与 18 比较
140 JBE     readfast  ; CL <= 18 则跳转至 readfast
141 MOV     CL,1
142 ADD     DH,1
143 CMP     DH,2
144 JB     readfast  ; DH < 2 则跳转至 readfast
145 MOV     DH,0
146 ADD     CH,1
147 JMP     readfast
```

程序 C-3　读取磁盘数据到内存

C.2 内存管理程序代码(memory.c)节选

```
/* 找到了足够大的内存 */
a = man->free[i].addr;
man->free[i].addr += size;
man->free[i].size -= size;
if (man->free[i].size == 0) {
  /* 如果 free[i] 变成了 0,就减掉一条可用信息 */
  man->frees--;
  for (; i < man->frees; i++) {
    man->free[i] = man->free[i + 1]; /* 代入结构体 */
  }
}
```

程序 C-4　分配内存

```
for (i = 0; i < man->frees; i++) {
  if (man->free[i].addr > addr) {
    break;
  }
}
```

程序 C-5　确定采取何种方式释放内存

```
98  /* 前面有可用内存 */
99  if (man->free[i - 1].addr + man->free[i - 1].size == addr) {
100   /* 可以与前面的可用内存归纳到一起 */
101   man->free[i - 1].size += size;
102   if (i < man->frees) {
103     /* 后面也有 */
104     if (addr + size == man->free[i].addr) {
105       /* 也可以与后面的可用内存归纳到一起 */
106       man->free[i - 1].size += man->free[i].size;
107       /* man->free[i] 删除 */
108       /* free[i] 变成 0 后归纳到前面去 */
109       man->frees--;
110       for (; i < man->frees; i++) {
111         man->free[i] = man->free[i + 1]; /* 结构体赋值 */
112       }
113     }
114   }
115   return 0; /* 成功完成 */
116 }
```

程序 C-6　前端可用

```
118 /* 不能与前面的可用空间归纳到一起 */
119 if (i < man->frees) {
120   /* 后面还有 */
121   if (addr + size == man->free[i].addr) {
122     /* 可以与后面的内容归纳到一起 */
123     man->free[i].addr = addr;
124     man->free[i].size += size;
125     return 0; /* 成功完成 */
126   }
127 }
```

程序 C-7　后端空闲

```
128 /* 既不能与前面归纳到一起,也不能与后面归纳到一起 */
129 if (man->frees < MEMMAN_FREES) {
130   /* free[i] 之后的,向后移动,腾出一点可用空间 */
131   for (j = man->frees; j > i; j--) {
132     man->free[j] = man->free[j - 1];
133   }
134   man->frees++;
135   if (man->maxfrees < man->frees) {
136     man->maxfrees = man->frees; /* 更新最大值 */
137   }
138   man->free[i].addr = addr;
139   man->free[i].size = size;
140   return 0; /* 成功完成 */
141 }
```

程序 C-8　前端后端均被占用

```
151 i = fifo32_get(&fifo);

161 if (256 <= i && i <= 511) { /* 键盘数据 */
162   if (i < 0x80 + 256) { /* 将按键编码转换为字符编码 */

171 if ('A' <= s[0] && s[0] <= 'Z') {  /* 当输入字符为英文字母时 */

177 if (s[0] != 0 && key_win != 0) { /* 一般字符、退格键、回车键 */

180 if (i == 256 + 0x0f && key_win != 0) {  /* Tab */

189 if (i == 256 + 0x2a) {  /* 左 Shift ON */
190   key_shift |= 1;
191 }
192 if (i == 256 + 0x36) {  /* 右 Shift ON */
193   key_shift |= 2;
194 }
195 if (i == 256 + 0xaa) { /* 左 Shift OFF */
196   key_shift &= ~1;
197 }
198 if (i == 256 + 0xb6) { /* 右 Shift OFF */
199   key_shift &= ~2;
200 }
201 if (i == 256 + 0x3a) {  /* CapsLock */

206 if (i == 256 + 0x45) {  /* NumLock */

211 if (i == 256 + 0x46) {  /* ScrollLock */

216 if (i == 256 + 0x3b && key_shift != 0 && key_win != 0) {  /* Shift+F1 */

227 if (i == 256 + 0x3c && key_shift != 0) {  /* Shift+F2 */

237 if (i == 256 + 0x57) {  /* F11 */
238   sheet_updown(shtctl->sheets[1], shtctl->top - 1);
239 }
240 if (i == 256 + 0xfa) {  /* 键盘成功接收到数据 */
241   keycmd_wait = -1;
242 }
```

```
247 } else if (512 <= i && i <= 767) { /* 鼠标数据 */
248   if (mouse_decode(&mdec, i - 512) != 0) {
249     /* 已经收集了 3 字节的数据,移动光标 */
250     mx += mdec.x;
251     my += mdec.y;
252     if (mx < 0) {
253       mx = 0;
254     }
255     if (my < 0) {
256       my = 0;
257     }
258     if (mx > binfo->scrnx - 1) {
259       mx = binfo->scrnx - 1;
260     }
261     if (my > binfo->scrny - 1) {
262       my = binfo->scrny - 1;
263     }
264     new_mx = mx;
265     new_my = my;
266     if ((mdec.btn & 0x01) != 0) {  /* 按下左键 */
```

程序 C-10　鼠标输入

```
185 void task_switch(void)
186 {
187   struct TASKLEVEL *tl = &taskctl->level[taskctl->now_lv];
188   struct TASK *new_task, *now_task = tl->tasks[tl->now];
189   tl->now++;
190   if (tl->now == tl->running) {
191     tl->now = 0;
192   }
193   if (taskctl->lv_change != 0) {
194     task_switchsub();
195     tl = &taskctl->level[taskctl->now_lv];
196   }
197   new_task = tl->tasks[tl->now];
198   timer_settime(task_timer, new_task->priority);
199   if (new_task != now_task) {
200     farjmp(0, new_task->sel);
201   }
202   return;
203 }
```

程序 C-11　多任务切换