第 2 章：操作系统结构

第 2 章：操作系统结构

- 操作系统服务 用户和操作系统接
- 口 系统调用
-
- 操作系统示例 系统程序
-
- 链接器和加载器 操作系统设计与实现
- 操作系统结构
-

目标

- 识别操作系统提供的服务 说明如何使用系统调用来提供操作系统服务
- 比较和对比用于设计操作系统的单片、分层、微内核、模块化和混合策略
-

操作系统服务

- 操作系统提供了一个环境
  - 程序执行 为程序和用户提供的其
  - 他服务
- 一组操作系统服务 - 对用户有帮助的操作系统功能：
  - 用户界面 - 几乎所有操作系统都有用户界面 (UI) - 命令行 (CLI)、图形用户界面 (GUI)、触摸屏、程序执行 - 系统必须能够将程序加载到
  - 内存中并运行该程序正常或异常结束执行（指示错误）
  - I/O 操作 - 正在运行的程序可能需要 I/O，这可能涉及文件或 I/O 设备

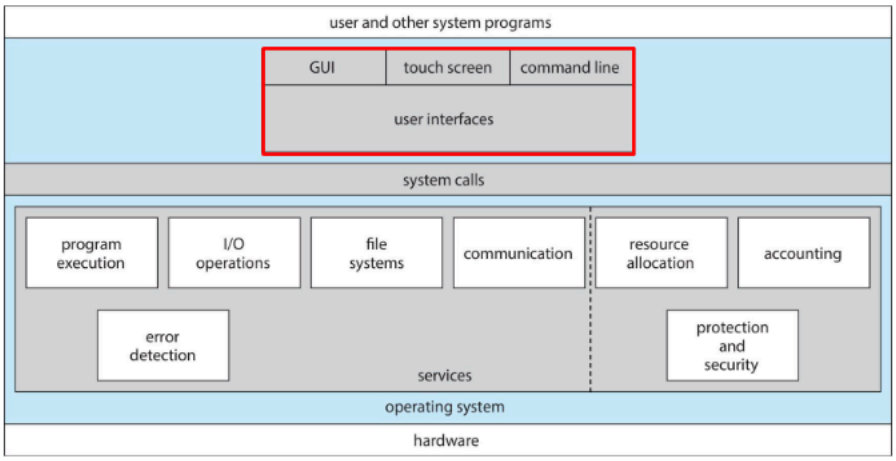
操作系统服务（续）

- 文件系统操作——文件系统特别重要。程序需要读写文件和目录、搜索、创建和删除它们、列出文件信息、权限管理。
- 通信——进程可以在同一台计算机上或通过网络在计算机之间交换信息
  - 通信可以通过共享内存或通过消息传递（由操作系统移动的数据包）
- 错误检测——操作系统需要不断意识到可能发生的错误 可能发生在CPU和内存、I/O设备、用户程序中 对于每种类型的错误，操作系统必须采取适当的措施来确保计算的正确性和一致性
  - 调试工具可以极大地提高用户和程序员有效使用系统的能力

操作系统服务（续）

- 另一套操作系统服务——通过资源共享保证系统本身高效运行的操作系统功能
- 资源分配 - 当多个用户或多个作业同时运行时，必须为每个用户分配资源
- 许多类型的资源 - CPU 周期、内存、文件存储、I/O 设备。日志记录 - 跟踪哪些用户使用了多少计算机资源以及何种类型的计算机资源
- 保护和安全 - 存储在多用户或网络系统中的信息的所有者可能希望控制该信息的使用，并发进程不应相互干扰
  - 保护涉及确保对系统资源的所有访问均受到控制 系统免受外部人员侵害的安全性需要用户身份验证，并扩展到保护外部 I/O 设备免受无效访问尝试

操作系统服务视图



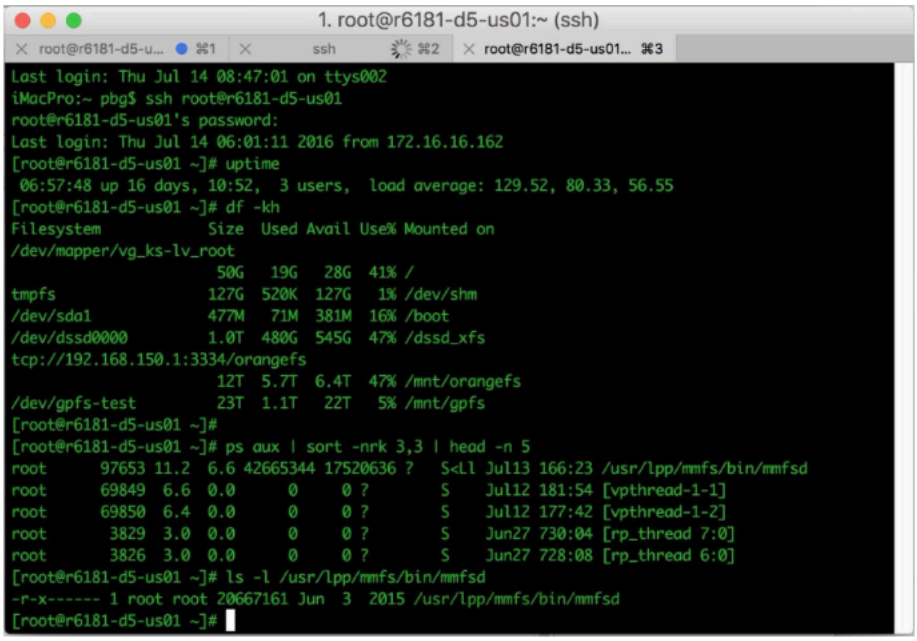
用户操作系统界面 - CLI

- 一般有三种做法。一种提供命令行界面（CLI）或命令解释器，允许用户直接输入要由操作系统执行的命令。另外两个允许用户通过图形用户界面或 GUI 和触摸屏与操作系统交互

CLI 或命令解释器允许直接输入命令

- 有时在内核中实现，有时由系统程序实现 有时实现多种风格 - Unix 或 Linux 中的 shell shell 的主要功能是 (1) 从用户获取命令，(2) 解释它并 (3) 执行它
- UNIX和Linux系统提供了不同版本的shell，例如C shell、Bourne-Again shell、Korn shell等

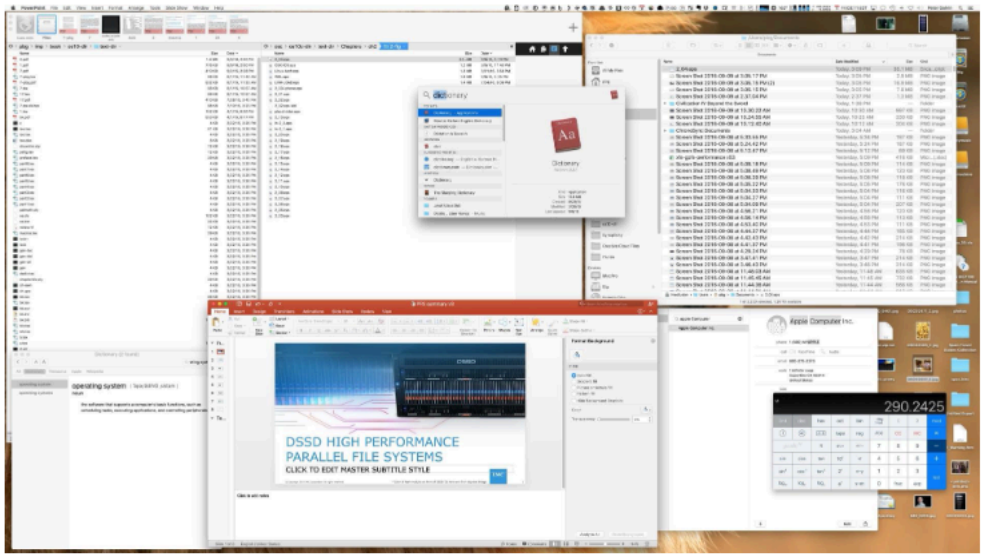
Bourne Shell 命令解释器



用户操作系统界面 - GUI

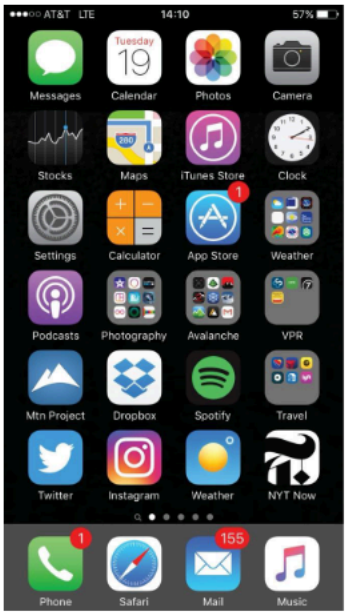
- 用户友好的桌面隐喻界面
  - 通常鼠标、键盘和显示器图标代表文件、程序、目录和系统功能界面中对象上的各种鼠标按钮会导致各种操作（提供信息、选项、执行功能、打开目录（称为文件夹））
  - 20 世纪 70 年代初由 Xerox PARC 发明，首次广泛应用于 Apple Macintosh（第一台 Mac 于 1984 年发布，取代了 70 年代的 Apple II）
- 现在许多系统都提供 CLI 和 GUI 界面
  - Microsoft Windows 使用 GUI 和 CLI “命令” shell Apple Mac OS X 是 “Aqua” GUI 界面，底层有 UNIX 内核和可用的 shell
  - Unix 和 Linux 具有带有可选 GUI 界面的 CLI（KDE、GNOME）

Mac OS X 图形用户界面

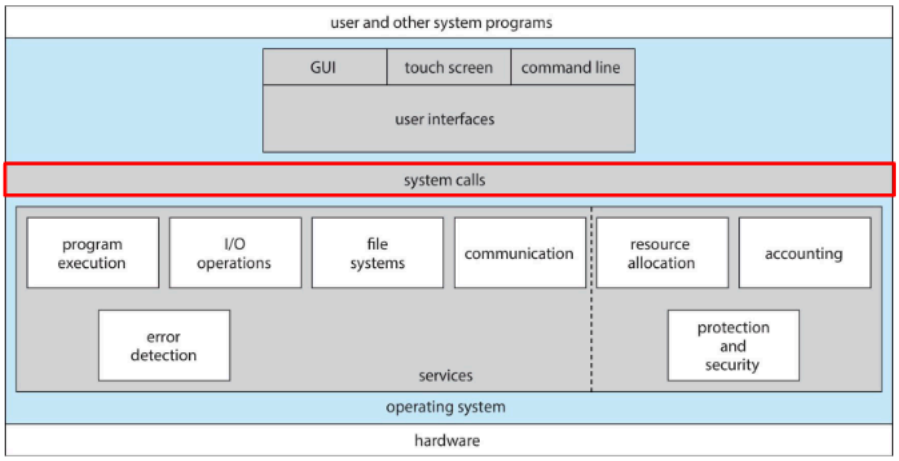


触摸屏界面

- 触摸屏设备需要新的界面
  - 鼠标不可行或不需要 基于手势的操作
  - 和选择 用于文本的虚拟键盘
  -
- 语音指令
  - iPhone Siri

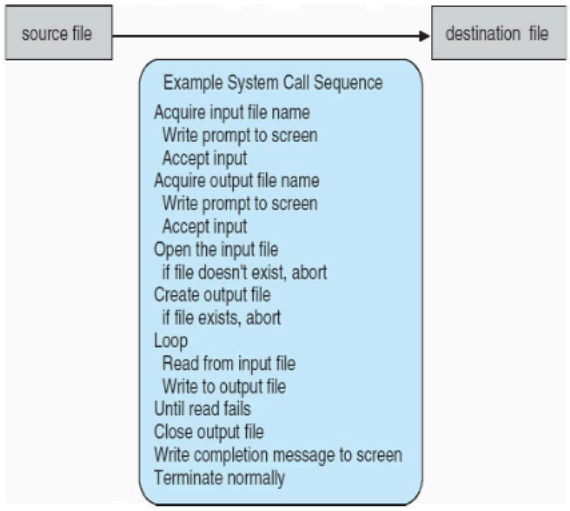


系统调用



系统调用（续）

- 操作系统提供的服务的编程接口
- 系统调用通常可用作以高级语言 (C/C++) 编写的函数，某些低级任务以汇编语言编写（访问硬件）
- 示例：cp in.txt out.txt – 涉及一系列系统调用



将一个文件复制到另一个文件

系统调用 - API

- 应用程序编程接口（API）指定了一组可供应用程序程序员使用的函数，包括传递给函数的参数和它可能期望的返回值
- 组成 API 的函数通常代表应用程序程序员调用实际的系统调用
  - 例如，Windows函数CreateProcess()调用Windows内核中的NTCreateProcess()系统调用
- 程序员通过操作系统提供的库访问 API
  - 对于 UNIX 和 Linux 中用 C 语言编写的程序，该库称为 libc
- 三种最常见的编程 API
  - 用于 Windows 系统的 Win32 API 用于基于 POSIX 的系统的
  - POSIX API（包括几乎所有版本的 UNIX、Linux 和 Mac OS X）
  - Java 虚拟机 (JVM) 的 Java API

标准API示例

Linux提供的库头文件

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value　function name　parameters

参数 • int fd – 要读取的文件描述符 • void *buf – 将读入数据的缓冲区 • size_t count – 要读入的最大字节数

缓冲区

返回值 • 0 表示文件结束 • -1 表示发生错误

系统调用 – API（续）

优点 - 为什么使用 API 而不是直接调用系统调用?

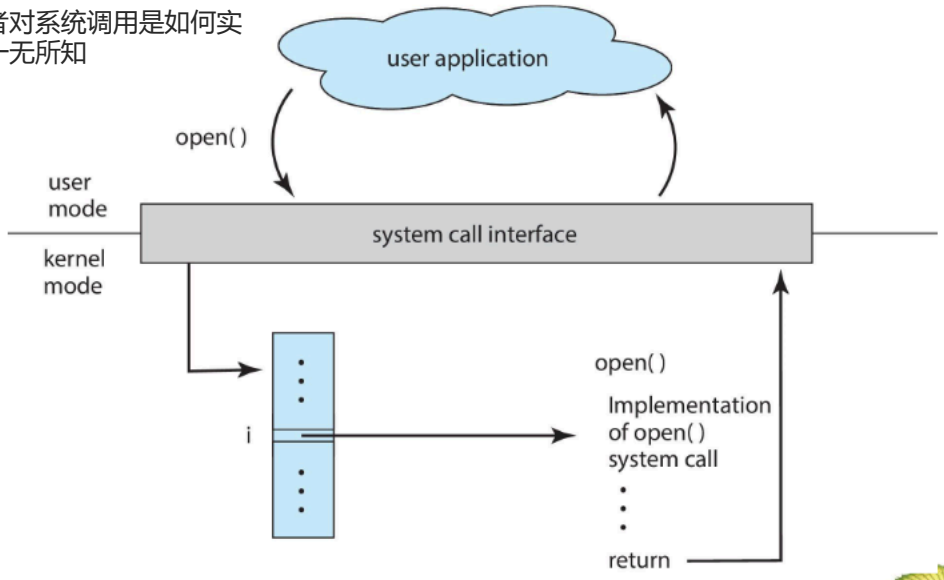
- 程序可移植性 – 使用 API 的程序员希望程序能够在任何支持相同 API 的系统上编译和运行 – 系统调用的实现因机器而异 对用户隐藏系统调用的复杂细节
-
  - 实际的系统调用通常比应用程序程序员可用的 API 更详细且更难以使用
  - API 的调用者（例如，程序）不需要知道系统调用是如何实现的或者它在执行期间做什么。相反，调用者只需要遵守 API（格式）并了解操作系统执行该系统调用后将执行的操作

系统调用实现

- 对于大多数编程语言，运行时环境 (RTE)（一组内置于库中的函数）提供系统调用接口，充当操作系统提供的系统调用的链接
  - 每个系统调用都关联一个身份号码。系统调用接口维护一个根
  - 据这些号码索引的表
- 系统调用接口 - 函数拦截 API 中的函数调用，调用操作系统内必要的系统调用，并返回系统调用的状态和返回值（如果有）

API – 系统调用 – 操作系统关系

调用者对系统调用是如何实现的一无所知

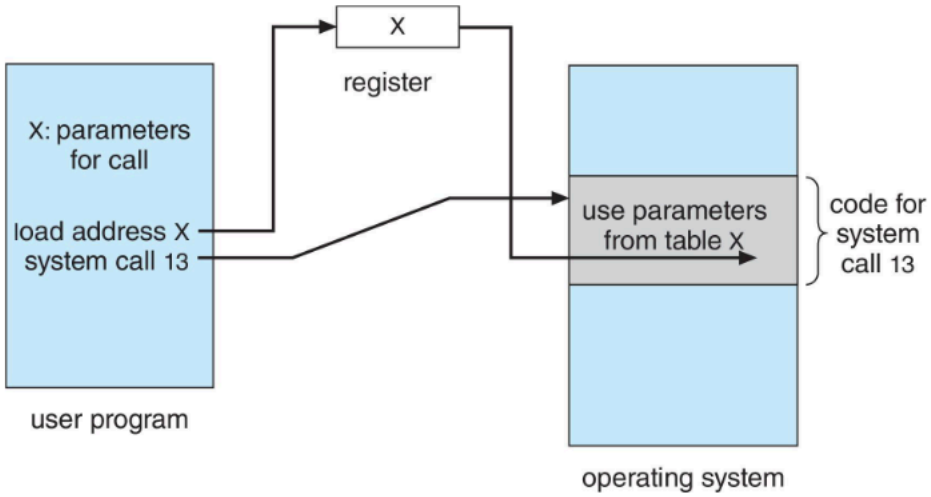


系统调用参数传递

- 通常，需要的信息比所需系统调用的标识更多
- 确切的类型和信息量根据操作系统和调用的不同而不同 三种通用方法用于将参数从用户程序传递到操作系统
  - 最简单 – 在寄存器中传递参数 □ 在某些情况下，参数可能比寄存器还多 块方法 – 参数存储在内存中的块或表中，块的地址作为参数传递到寄存器中
  - 堆栈方法——由程序将参数放置或压入堆栈并由操作系统从堆栈中弹出

块和堆栈方法不限制传递参数的数量或长度

通过表传递参数



系统调用的类型

- 过程控制
  - 创建进程、终止进程、结束、
  - 中止
  - 加载、执行获取进程属性、设置进程
  - 属性等待时间
  -
  - 等待事件、信号事件 分配和释放内存 如果出现错误则
  - 转储内存 用于确定错误的调试器、单步执行 用于管理
  - 进程间共享数据访问的锁
  -
  -

系统调用的类型（续）

- 文件管理
  - 创建文件、删除文件
  - 打开、关闭文件 读取、写入、重新定位
  - 获取和设置文件属性
  -
- 设备管理
  - 请求设备、释放设备 读取、写入、
  - 重新定位 获取设备属性、设置设备属性 逻辑连接或分离设备
  -

系统调用的类型（续）

- 信息维护
  - 获取时间或日期、设置时间或日期 获取系统数据、设置系统数据 获取和设置进程、文件或设备属性
  -
- 通讯
  - 创建、删除通信连接 发送、接收消息（如果消息传递模型为主机名或进程名）
  - 共享内存模型创建并访问内存区域传输状态信息
  -

  连接和分离远程设备保护
-
  - 控制对资源的访问，获取和设置权限允许和拒绝用户访问

Windows 和 Unix 系统调用示例

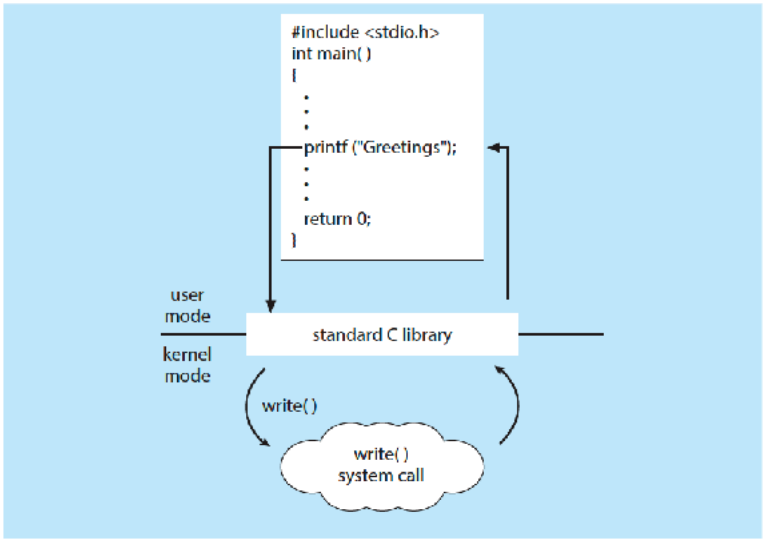
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

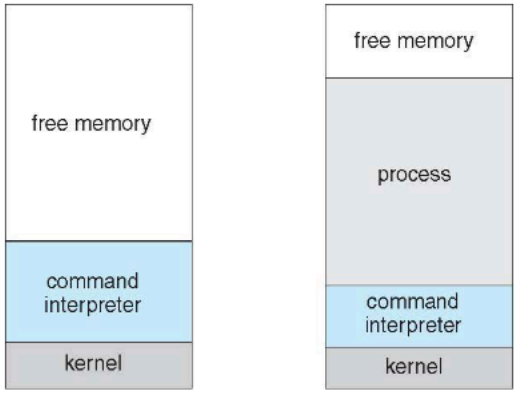
标准 C 库示例

- C 程序调用 printf() 库调用，该调用调用 write() 系统调用



示例：MS-DOS

- 单任务
- 系统启动时调用 Shell
- 运行程序简单 无需创建进程
- 单一内存空间将程序加载到内存中，覆盖除内核之外的所有内容
- 程序退出 -> shell 重新加载



系统启动时　　运行一个程序

系统程序

- 现代计算机系统的另一个方面是其系统服务的集合
- 系统服务，或者说系统程序，或者说系统实用程序，为程序开发和执行提供了便利的环境
  - 其中一些只是系统调用的用户界面；其他的要复杂得多
- 大多数用户看到的操作系统的视图是由应用程序和系统程序定义的，而不是实际的系统调用
  - 考虑具有鼠标和 Windows 界面的 GUI 以及命令行 UNIX shell。两者本质上都使用相同的系统调用集，但系统调用的外观非常不同，并且以不同的方式起作用 – 用户视图

系统程序（续）

- 文件管理
  - 创建、删除、复制、重命名、打印、转储、列出以及一般操作文件和目录
- 状态信息
  - 有些询问系统日期、时间、可用内存量、磁盘空间、用户数量
  - 其他提供详细的性能、日志记录和调试信息
- 文件修改
  - 用于创建和修改文件的文本编辑器 用于搜索文件内容或执行文本转换的特殊命令

系统程序（续）

- 编程语言支持
  - 通常提供常见编程语言（例如 C、C++、Java 和 Python）的编译器、汇编器、调试器和解释器
- 程序加载和执行
  - 绝对加载器、可重定位加载器、链接编辑器和覆盖加载器、高级和机器语言的调试系统
- 通信——提供在进程、用户和计算机系统之间创建虚拟连接的机制
  - 允许用户向彼此的屏幕发送消息、浏览网页、发送电子邮件、远程登录、将文件从一台计算机传输到另一台计算机

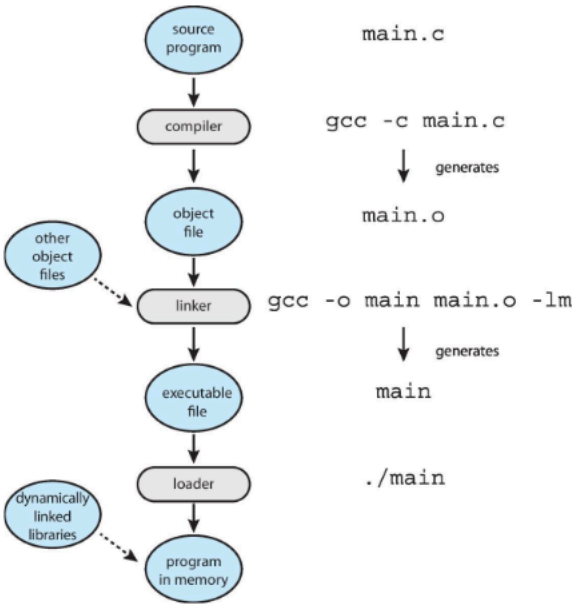
系统程序（续）

- 后台服务
  - 在启动时启动
    - 其中一些进程在完成任务后终止 一些进程继续运行直到系统停止，通常称为
    - 服务、子系统或守护进程 提供磁盘检查、进程调度、错误日志记录等设施
- 应用程序
  - 不是操作系统的一部分 通过命令行、鼠标单击、手指点击启动 示例包括网络浏览器、文字处理器、文本格式化程序、电子表格、数据库系统和游戏。

链接器和加载器

- 源代码被编译成目标文件，旨在加载到任何物理内存位置 - 可重定位目标文件
- 链接器将这些目标文件组合成一个二进制可执行文件，如果需要的话还可以包含库
- 程序作为二进制可执行文件驻留在辅助存储上，必须由加载程序将其加载到内存中才能执行
  - 重定位将最终地址分配给程序部分，并调整程序中的代码和数据以匹配这些地址（稍后讨论）
- 现代通用操作系统不会将库静态链接到可执行文件中
  - 相反，动态链接库（在 Windows 中为 DLL）在需要时加载，这些库由使用同一库的相同版本的所有程序共享（仅加载一次）
- 目标文件、可执行文件具有标准格式（机器代码和符号表），因此操作系统知道如何加载和启动它们

链接器和加载器的作用



保护

- 保护是一个内部问题，相反，安全必须考虑系统和系统使用的环境。操作系统必须保护自身免受用户程序的侵害
-
  - 可靠性：损害操作系统通常会导致其崩溃 安全性：限制进程可以执行的操作范围
  - 隐私：将每个进程限制为允许访问的数据 公平性：每个进程都应限制其适当的系统资源份额
- 系统保护功能以需要了解的原则为指导，并实施机制来强制执行最小权限原则。计算机系统包含必须防止误用的对象。对象可以是硬件（例如内存、CPU 时间和 I/O 设备）或软件（例如文件、程序和信号量）。

操作系统设计与实现

操作系统的设计和实现不是“可解决的”，但一些方法已被证明是成功的

- 操作系统的内部结构差异很大 在高层，设计受到硬件选择、系统类型
- （例如台式机/笔记本电脑、移动设备、分布式系统或实时系统）的影响 通过定义目标和系统类型来开始设计规格
-
  - 用户目标：操作系统应该使用方便、易学易用、可靠、安全、快速
  - 系统目标：操作系统应易于设计、实现和维护，并且灵活、可靠、无差错、高效
- 指定和设计操作系统是软件工程中一项极具创造性的任务

操作系统设计与实现（续）

- 分离政策的重要原则：将做什么？机制：怎么做？机制指定如何做事；政策决定将做什么
-
  - 计时器构造是一种确保 CPU 保护的机制，但决定为特定用户设置计时器多长时间是一个策略决策
- 政策与机制分离是一个非常重要的原则，如果政策决定发生变化，它可以提供灵活性，而且确实如此
  - 如果正确分离，它可以用于支持 I/O 密集型程序优先于 CPU 密集型程序的策略决策，也可以用于支持相反的策略。
- 简而言之，操作系统的设计考虑了特定的目标。这些目标最终决定了操作系统的策略。操作系统通过特定的机制来实现这些策略。

执行

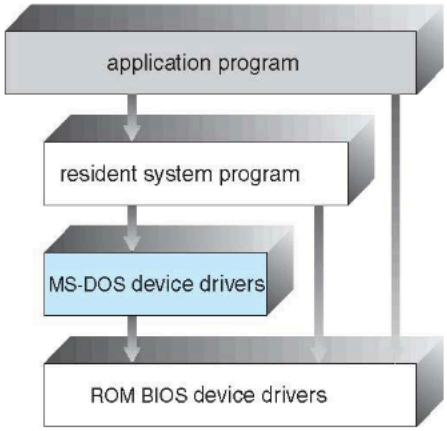
- 早期的操作系统是用汇编语言编写的，现在，大多数操作系统都是用高级语言（例如 C 或 C++）编写的
  - 内核的最低级别可能仍然是汇编语言。经常使用不止一种高级语言。大多数Android系统库都是用C或C++编写的，其应用框架大多用Java编写 使用高级语言的优点是更容易移植到其他硬件
  - 代码可以写得更快、更紧凑、更容易理解和调试
- 唯一可能的缺点是速度降低和存储要求增加——这不是当今计算机系统的主要问题

操作系统结构

- 通用操作系统是一个非常大的程序，有多种构造方法
  - 整体结构 分层 – 一种特定类型的模块化
  - 方法 微内核 – Mach OS
  - 可加载内核模块或 LKM – 模块化方法

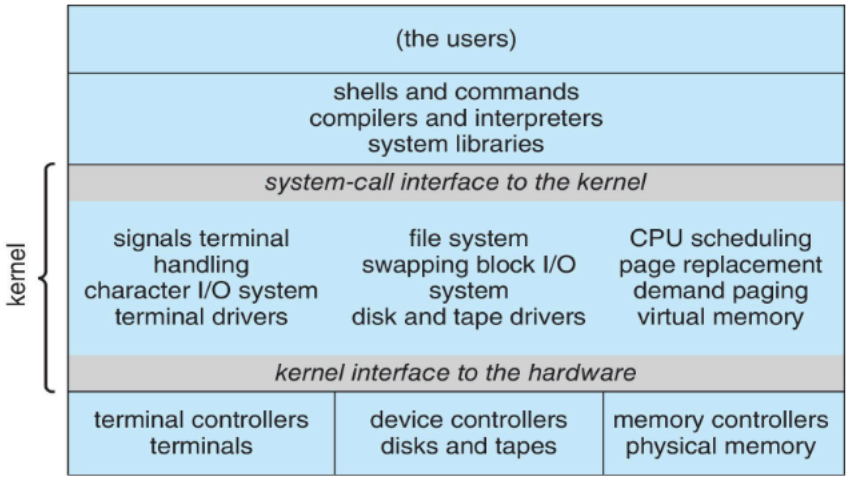
结构简单

- 此类操作系统没有明确定义的结构，通常以小型、简单且有限的系统开始
- MS-DOS – 旨在以最小的空间提供最多的功能
  - 没有仔细划分模块 虽然它有一定的结构，但接口和功能级别没有很好地分开 - 即应用程序可以直接访问 I/O 为 Intel 8088 编写，没有双模式（内核模式与用户模式）并且没有硬件保护



整体结构——原始 UNIX

- 在传统的 UNIX 中，内核由系统调用接口以下和物理硬件之上的所有内容组成

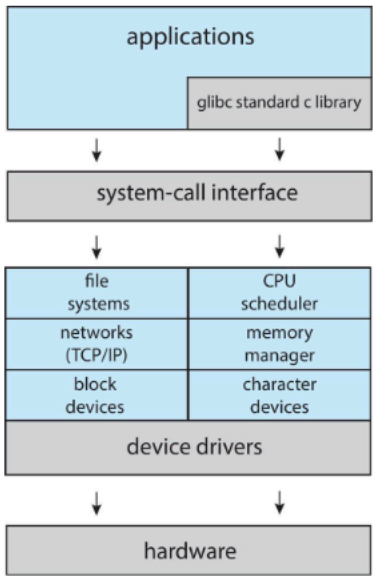


整体结构——原始 UNIX

- UNIX – 最初受到硬件功能的限制，最初的 UNIX 操作系统的结构有限
- 将内核的所有功能放入单个静态二进制文件中，该文件在单个地址空间中运行 - 称为整体结构
- UNIX 由两个独立的部分组成：内核和系统程序
  - 内核进一步分为一系列接口和设备驱动程序，这些年来随着 UNIX 的发展，这些接口和设备驱动程序已经得到了相当大的扩展
- 缺点是大量的功能集中在一个层次上，导致实现、调试和维护困难
- 单片内核具有明显的性能优势，因为系统调用接口的开销非常小，并且内核内的通信速度很快
  - 速度和效率 – 仍在 UNIX、Linux 和 Windows 中使用

Linux系统结构

- Linux 操作系统基于 UNIX，结构类似
- Linux 内核是整体式的，因为它也完全在单个地址空间中以内核模式运行
- 应用程序在与内核的系统调用接口通信时使用glibc标准C库它具有模块化设计，允许在运行时修改内核（待讨论的LKM）



模块化设计

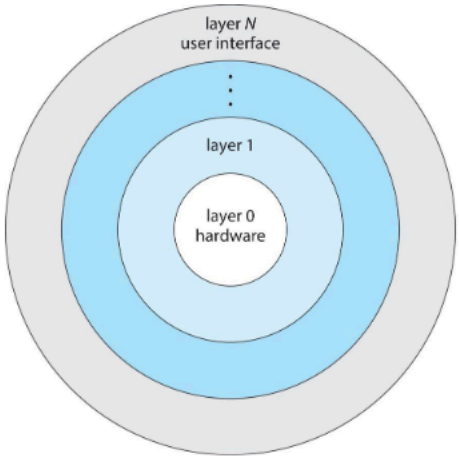
- 整体方法通常被称为紧密耦合系统，因为系统某一部分的更改会影响其他部分——本质上是一个在一个地址空间中运行的大程序（将在第 3-4 章中讨论）设计一个松散耦合的系统，该系统分为分成独立的、较小的组件，每个组件都有特定且有限的功能
-

模块化方法的优点是，一个组件的更改仅影响该组件，而不会影响其他组件，从而在创建和修改每个组件时具有更大的自由度和灵活性。系统可以通过多种方式实现模块化。一种方法是分层方法

-

分层方法

- 操作系统可以分为多个层，每个层都构建在较低层之上。最底层（第0层）是硬件；最高层（N 层）是用户界面。
- 每一层都由数据结构和一组函数组成，这些函数可以被高层调用，反过来，也可以调用低层的操作。简而言之，每一层都利用来自低层的服务（如果有的话），提供一组功能，并为更高层提供某些服务（如果有）



分层方法（续）

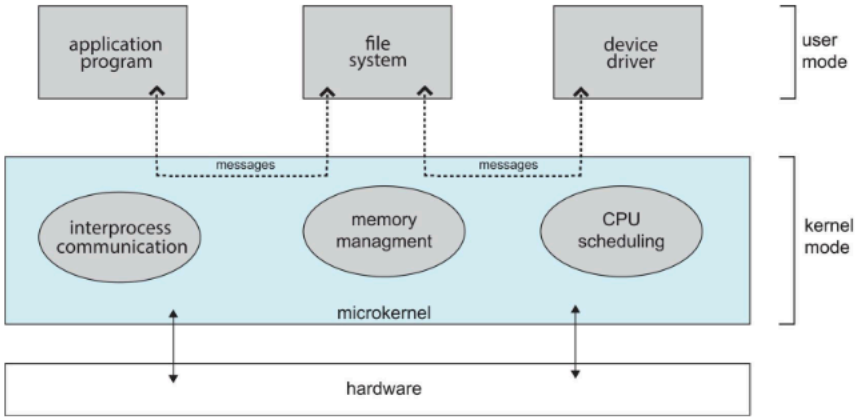
- 信息隐藏：一层不需要知道下层操作是如何实现的，只需要知道这些操作做了什么——接口
  - 每层都向更高层隐藏其自身的数据结构、操作和硬件的存在
- 分层方法的主要优点是简化构建和调试——从最低层开始
  - 分层系统已成功用于许多其他软件系统，例如计算机网络（例如 TCP/IP/链路物理）和 Web 应用程序
- 然而，很少有操作系统使用纯粹的分层方法，因为
  - 适当定义不同层及其各自的功能存在重大挑战
  - 由于需要程序遍历多层来获取服务的开销，整体性能可能会受到影响
- 趋势是层数更少、功能更多、代码模块化，同时避免复杂的层定义和交互问题

微内核系统结构

- 内核变得很大并且难以管理从内核中删除所有非必要的组件并将
- 它们实现为单独地址空间中的系统或用户级程序这导致内核变得更小
-
- Mach 于 20 世纪 80 年代中期在 CMU 开发，使用微内核方法对内核进行模块化
  - 最著名的微内核操作系统是 Darwin，用于 Mac OS X 和 iOS。它由两个内核组成，其中之一是Mach微内核
- 关于哪些服务应保留在微内核中以及哪些服务应在用户空间中实现，几乎没有达成共识，但通常，除了通信设施外，微内核还提供最少的进程和内存管理

微内核系统结构（续）

- 微内核的主要功能之一是通过消息传递的方式提供程序与运行在用户地址空间中的各种服务之间的通信。例如，如果应用程序希望访问文件，则它必须与文件服务器进行交互。程序和服务从不直接交互。相反，它们通过与微内核交换消息来间接通信。



微内核系统结构（续）

- 优点：
  - 更容易扩展微内核操作系统，因为所有新服务通常都添加到用户空间而无需修改内核
  - 当必须修改内核时，更改较少，因为内核要小得多
  - 更容易将操作系统移植到新架构（硬件）更安全、更可靠（在内核模式下运行的代码更少），因为大多数服务都作为用户进程运行，而不是内核进程。如果服务失败，操作系统的其余部分保持不变
- 缺点：
  - 由于系统功能开销、用户空间到内核空间通信的增加，微内核的性能可能会受到很大影响
  - 当两个用户级服务必须通信时，必须在驻留在不同地址空间的服务之间复制消息
- Window NT 具有分层微内核，性能比 Window 95 差，与 Window XP 相比更加单一（将功能移至内核）

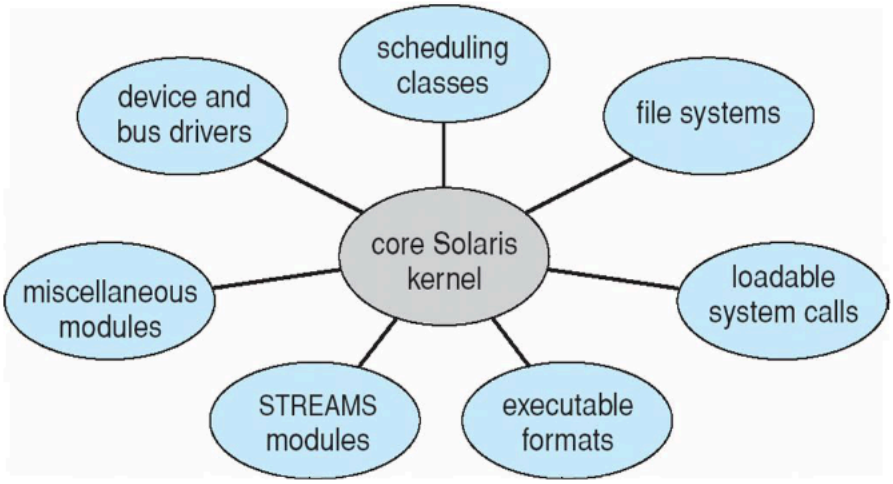
模块化方法

- 当前操作系统设计中最好的方法涉及使用可加载内核模块或 LKM
- 内核有一组核心组件，可以在启动时或运行时通过模块链接附加服务。
- 主要思想是让内核提供核心服务，而其他服务则动态实现和添加，当内核运行时动态链接服务比直接向内核添加新功能更好，因为每次更改都需要重新编译整个内核被做了
  - 例如，内核将CPU调度和内存管理算法放入内核中，然后通过可加载模块的方式添加对不同文件系统的支持

模块化方法（续）

- 这类似于分层设计，因为每个内核部分都有一个定义良好、受保护的接口，但它更灵活，因为任何模块都可以调用任何其他模块
- 它还类似于微内核，因为主模块仅具有核心功能以及如何加载其他模块和与其他模块通信的知识；但这比微内核设计更有效，因为模块不需要调用消息传递来进行通信 Linux 使用可加载内核模块，主要用于支持设备驱动程序和文件系统。
- LKM 可以在系统启动时或运行时“插入”内核，也可以在运行时从内核中删除
  - 例如，将 USB 设备插入正在运行的机器中。如果Linux内核没有必要的驱动程序，可以动态加载。
- 这允许动态和模块化内核，同时保持整体系统的性能优势以提高效率

Solaris 模块化方法



混合系统

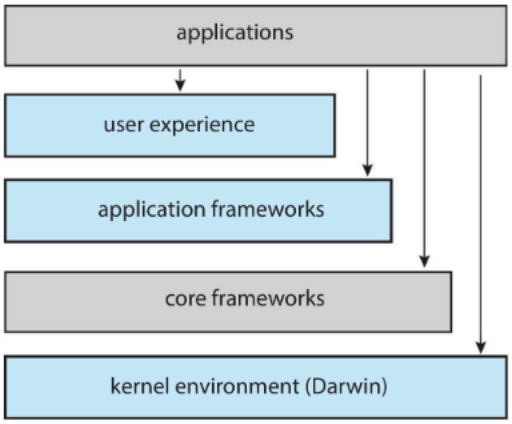
- 实际上，很少有操作系统采用单一的、严格定义的结构。相反，它们结合了不同的结构，形成了混合系统。混合系统结合了多种方法来满足性能、安全性和可用性需求
- Linux 是整体式的，因为单个地址空间中的操作系统提供了高效的性能。它也是模块化的，因此可以动态地将新功能添加到内核中
- Windows 很大程度上是整体式的，但它通过为作为用户模式进程运行的单独子系统（称为个性）提供支持，保留了微内核系统的一些典型行为。Windows系统还提供对动态可加载内核模块的支持

操作系统设计总结

- 单一操作系统没有结构；所有功能都在单个静态二进制文件中提供，该文件在单个地址空间中运行。尽管此类系统很难修改，但它们的主要好处是效率。分层操作系统分为许多离散的层，其中底层是硬件接口，最高层是用户接口。尽管分层软件系统已经取得了一些成功，但由于性能问题和定义适当层的困难，它通常对于设计操作系统来说并不理想。设计操作系统的微内核方法使用最小内核；大多数服务作为用户级应用程序运行，其中通过消息传递进行通信
- 用于设计操作系统的可加载内核模块方法通过可以在运行时加载和删除的模块提供操作系统服务。
- 许多现代操作系统都是使用整体内核和模块组合构建的混合系统。

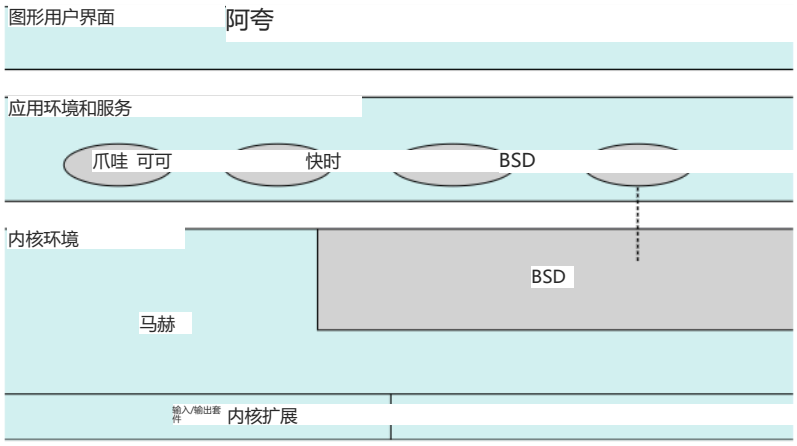
macOS 和 iOS 结构

- 在架构上，macOS（适用于台式机和笔记本电脑）和 iOS（iPhone 和 iPad）有很多共同点 – 用户界面、编程（语言）支持、图形和媒体以及内核环境 – Darwin 包括 Mach 微内核和 BSD UNIX 内核



Mac OS X 结构

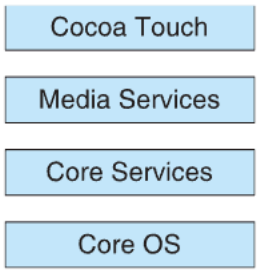
- Apple Mac OS X 是混合、分层的 Aqua UI（用于鼠标或触控板），加上为 Objective-C 提供 API 的 Cocoa 编程环境
- 核心框架。支持图形和媒体，包括 Quicktime 和 OpenGL 内核环境，也称为 Darwin，包括 Mach 微内核和 BSD Unix



iOS系统

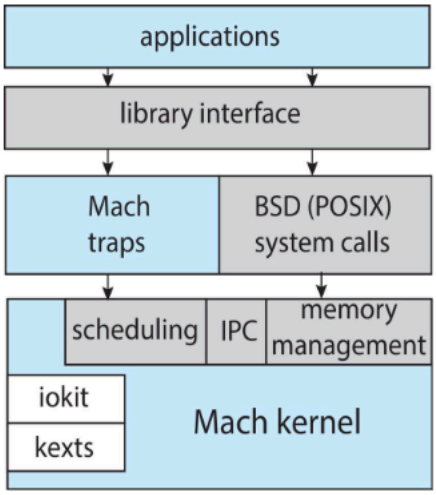
适用于 iPhone、iPad 的 Apple 移动操作系统

- 基于 Mac OS X 构建，增加了功能 本身不运行 Mac OS X 应用程序
  - 还可以在不同的 CPU 架构上运行（ARM 与 Intel）
- Springboard 用户界面，专为触摸设备设计。
- 用于在移动设备（触摸屏）上开发应用程序的 Cocoa Touch Objective-C API
- 图形、音频、视频的媒体服务层 – Quicktime、OpenGL
- 核心服务提供云计算、数据库核心操作系统，基于Mac OS X内核



达尔文

- 由 Mach 微内核和 BSD UNIX 内核组成的分层系统 – 混合系统
- 两个系统调用接口 - Mach 系统调用（称为陷阱）和 BSD 系统调用（提供 POSIX 功能） 该接口是一组丰富的库 - 标准 C 库以及支持网络、安全和编程语言的库
- Mach 提供基本的操作系统服务，包括内存管理、CPU 调度和 IPC 设施
- 内核环境为设备驱动程序和动态可加载模块（称为内核扩展或 kext）提供 I/O 工具包

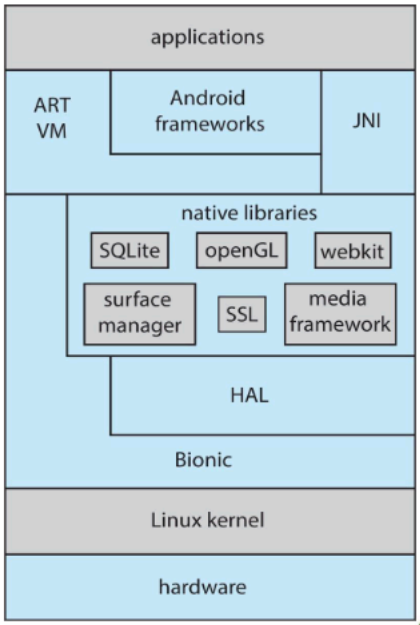


安卓

- 由开放手机联盟（主要由 Google 领导）开发 Android 运行在各种移动平台上，并且是开源的，相比之下 iOS 运行在 Apple 移动设备上，并且是闭源的 Android 与 iOS 类似 - 一个分层系统，提供支持图形、音频和硬件功能的丰富框架。 Google 没有使用标准的 Java API，而是为 Java 开发设计了单独的 Android API。 Java 应用程序在 Android RunTime 或 ART 上执行，这是一种针对内存和 CPU 处理能力有限的移动设备进行优化的虚拟机
- Java 本机接口或 JNI，它允许开发人员绕过虚拟机并编写可以访问特定硬件功能的 Java 程序。
  - 使用 JNI 编写的程序通常不能从一种硬件移植到另一种硬件。

安卓架构

- 这些库包括用于开发 Web 浏览器 (webkit)、数据库支持 (SQLite) 和网络支持（例如安全套接字 (SSL)）的框架。为了使Android能够在任何硬件设备上运行，硬件抽象层或HAL抽象了所有硬件，例如相机、GPS芯片和其他传感器，并为应用程序提供独立于特定硬件的一致视图 Google为Android开发了Bionic标准C库，而是使用适用于 Linux 系统的标准 GNU C 库 (glibc) 修改后的适用于移动系统的 Linux 内核，包括电源管理。



第 2 章结束