读书笔记 | 操作系统概念

2018-07-11 | Book | 0 | 阅读次数： 80
50k

大二下学期课程，记录一下学习的重点和笔记

《操作系统概念》这本书主要介绍关于操作系统的八个部分：操作系统的主要部件， 基本的计算机组成结构， 进程管理， 内存管理， 储存管理， 保护与安全， 分布式及系统， 专用系统。

这里的笔记主要包括了这本书的四大部分：概述、进程管理、内存管理和存储管理。

存储管理还有一部分没有完成，后面的保护与安全、分布式系统、特殊用途系统还有案例研究以后有时间再慢慢写

第一部分 概论

第 1 章 导论

操作系统是管理计算机硬件的程序， 为应用程序提供基础， 并且充当计算机硬件和计算机用户的中介。

计算机系统大致分为 4 个部分：

- 计算机硬件
- 操作系统
- 系统程序与应用程序
- 用户

计算机系统操作

计算机开始运行的时候，通常需要运行一个初始化程序 / 引导程序 (bootstrap program)，通常位于 ROM 或 EEPROM 中，称为计算机硬件中的固件。

它负责初始化系统中的所有部分，包括：CPU 寄存器， 设备控制器， 内存内容。它必须知道如果装入系统并开始执行系统，定位操作系统内核并把他装入内存，然后操作系统就开始初始化。

中断

事件的发生通常通过硬件或软件中断 (interrupt) 来表示，硬件可以随时通过系统总线向 CPU 发出信号，触发中断，而软件通过执行特别操作如系统调用(system call)/ 监视器调用(monitor call)

中断是计算机结构的重要组成部分。中断必须将控制转移到合适的中断处理程序，简单的方法就是调用一个通用的子程序检查中断信息，然后调用对应的中断处理程序。

也可以使用中断处理子程序的指针表，间接地调用，这样就不需要经过其他的中间子程序。通常指针表处于低地址内存，包括了各种设备的中断处理子程序的地址，地址的数组或中断向量 (interrupt vector) 可以通过唯一设备索引号来提供子程序地址。

中断程序必须负责恢复中断前的状态。

存储结构

典型指令执行周期

- 从内存中获取指令， 保存在指令寄存器 (instruction register)
- 指令被解码，并可能导致从内存获取操作数或将操作数保存在内部寄存器中
- 执行指令

理想情况下， 程序和数据都永久留在内存中，但是这是不可能的，因为

- 内存太小
- 内存是易失性存储设备

所以需要辅存 (secondary storage)， 如磁盘 (magnetic disk)， SSD

存储设备层次：(估计比较老的了)

- 寄存器
- 高速缓存
- 主存
- 电子磁盘
- 磁盘
- 光盘
- 磁带

I/O 结构

通用计算机系统由一个 CPU 和多个设备控制器组成。通过共同的总线连接起来，每个设备控制器负责特定类型的设备，可有多个设备与其相连。

SCSI(small computer system interface) 控制器可以有 7 个或更多的设备相连。

设备控制器

- 维护一定量的本地缓冲存储和一组特定用途的寄存器
- 复制在其控制的外部设备与本地缓冲存储之间进行数据传递
- 通常每个设备控制器都有一个设备驱动程序

I/O 中断驱动 (适合少量的数据)

- 设备驱动程序在设备控制器中装载适当的寄存器
- 设备控制器检查寄存器状态决定操作
- 控制器开始向本地缓冲区传输数据
- 设备控制器通过中断通知设备驱动程序已完成操作

- 设备驱动程序返回对系统的控制。

DMA(direct memory access) 直接内存访问

设备直接写入内存而不需要 CPU 的干预，每一块只产生一个中断。

交换使得各个部件并发对话而不是在总线上争夺事件，更加地高效

计算机系统体系结构

- 单处理器系统
  - 只有一个通用 CPU
  - 还包含其他特定目的微处理器， 用来克服主 CPU 超载问题
- 多处理器系统 (并行系统 Parallel system, 紧耦合系统 tightly coupled system)
  - 拥有多个紧密通信的 CPU，共享计算机总线，时钟，内存和外设

  优点：
  - 增加吞吐量
  - 规模经济
  - 增加可靠性

计算机系统不断增加的可靠性是很关键的，这种能提供与正常工作的硬件成正比的服务的能力称为适度退化 (graceful degradation)。超出适度退化的能力被称为容错 (fault tolerant)

多处理器系统主要分为

非对称多处理 (asymmetric multiprocessing)： 主从处理器

对称多处理 (symmetric multiprocessing， SMP)

概念： 刀片服务器 (blade server) 每个刀片处理器独立启动并运行各自的系统

集群系统：

由两个或多个独立的系统耦合起来的

通过局域网连接或更快的内部连接 (InfiniBand)

用途： 提供高可用性 (high availability) 服务

分类：对称与非对称

非对称集群 (asymmetric clustering) 中， 一台机器处于热备份状态(hot standby mode)，另一台运行程序。

对称集群 (symmetric clustering)，两个或多个主机都运行程序并互相监视。

还有并行集群和 WAN 集群。

并行集群中通常需要分布式锁管理器 (distributed lock manager， DLM)

操作系统结构

分时操作系统、多道程序设计

进程 (process): 装到内存并执行的程序

作业调度 (job scheduling)： 在储存在磁盘作业池(job pool) 中与主存中的作业做出决策和安排

CPU 调度 (CPU scheduling)： 多个任务同时执行

在分时操作系统中，为了保证合理的相应时间，就需要通过交换来得到。通常使用虚拟内存 (virtual memory) 实现。

操作系统操作

现代操作系统通常是由中断驱动的。

操作系统的合理设计必须确保错误程序 (或恶意程序) 不会造成其他程序执行错误。

双重模式操作

区分操作系统代码和用户自定义代码的执行，通常提供硬件支持以允许区分各种执行模式 (在计算机硬件增加一个模式位(mode bit) 来区分)

- 用户模式 (user mode)
- 内核模式：监督程序模式 (monitor mode)[也称为管理模式(supervisor mode)]、系统模式(system mode) 或特权模式 (privileged mode)

只要操作系统获得了对计算机的控制，就处于内核模式。

某些特权指令 (privileged instruction) 只能在内核模式执行。

特权指令：可以引起损害的机器指令，如切换到用户模式、I/O 的控制、定时器的管理和中断管理

系统调用通常采用陷阱到中断向量中的一个指定位置的方式。

定时器

确保操作系统维持对 CPU 的控制，防止用户程序陷入死循环不返回控制权，通常使用定时器 (timer) 定时中断。

进程管理

进程：处于执行中的程序

需要一定的资源

程序本身不是进程，程序时被动的实体。

单线程进程具有一个程序计数器来明确下一个执行的指令，这样程序的执行必须时连续的。

进程时系统工作的单元。系统由多个进程组成，其中一些事操作系统进程，其他是用户进程。

操作系统需要负责：

- 创建和删除用户进程和系统进程
- 挂起和重启进程
- 提供进程同步机制
- 提供进程通讯机制
- 提供死锁处理机制

内存管理

内存是现代计算机操作系统的中心，是 CPU 和 I/O 设备共同快速访问的数据仓库，通常也是 CPU 可以直接寻址和访问的唯一大容量存储器。

操作系统负责下列有关的内存管理的活动：

- 记录内存的哪部分正在被使用和被谁使用
- 当有内存空间是，决定哪些进程可以装入内存
- 根据需要分配和释放内存空间

存储管理

文件系统管理

操作系统负责：

- 创建和删除文件
- 创建和删除目录来组织文件
- 提供操作文件和目录的原语
- 将文件映射到二级存储上
- 在稳定存储介质上备份文件

大容量存储器管理

操作系统负责：

- 空闲空间管理
- 存储空间分配
- 硬盘调度

高速缓存

由于高速缓存的大小有限，因此高速缓存管理 (cache management) 的设计就显得十分地重要。

高速缓存的大小和置换策略的仔细选择可以极大地提高性能.

高速缓存需要解决高速缓存一致性 (cache coherency), 这通常是硬件问题。以确保每个进程都得到最近已更新的值

I/O 系统

操作系统的目的之一： 对用户隐藏具体硬件设备的特性。

而 I/O 子系统也对操作系统隐藏一些 I/O 设备的特性。只有设备驱动才知道特性。

保护和安全

保护： 一种控制进程或用户对计算机系统资源的访问的机制

安全： 防止系统不受外部或内部攻击

分布式系统

网络， 两个或多个系统之间的通讯路径

一个网络协议只简单地需要一个接口设备，加上管理他的驱动程序以及按网络协议处理数据的软件

根据节点间的距离来分，网络可以分为

- 局域网 (local-area network， LAN)
- 广域网 (wide-area network， WAN)
- 城域网 (metropolitan-area network，MAN)
- 小域网 (small-area network， SAN)

网络操作系统 (network operating system) 提供了比网络连接更进一步的网络和分布式系统的概念

专用系统

- 实时嵌入式系统
- 多媒体系统
- 手持系统

计算环境

传统计算

客户机 - 服务器计算

- 计算服务器系统
- 文件服务器系统

对等计算

对等 (P2P) 系统模式

基于 Web 的计算

第 2 章 操作系统结构

操作系统服务

操作系统提供给用户的常用函数：

- 用户界面 (user interface, UI)：
    - 命令行界面 (command-line interface, CLI)
    - 批界面 (batch interface)，不常见
    - 图形用户界面 (graphical user interface, GUI)
- 程序执行
- I/O 操作
- 文件系统操作
- 通信： 一个进程与另一个进程进行交换信息，可以通过共享内存或信息交换来实现
- 错误检测

确保系统高效运行的函数：

- 资源分配：
    - 如，CPU 周期， 内存，文件储存：需要特别的分配代码
    - 如， I/O 设备：只需要通用的请求和释放代码
- 统计 (accounting)
- 保护和安全

操作系统的用户界面

命令解释程序 (Command Interpreters)

获取并执行用户指定的下一条命令

有两种常用的方法:

- 命令解释程序本身包含代码以执行这些命令
- 由系统程序实现绝大多数命令，命令解释程序只需要调用

图形用户界面 (Graphical User Interfaces)

提供基于鼠标的窗口和菜单系统作为接口

- Usually mouse, keyboard, and monitor
- Icons represent files, programs, actions, etc
- Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Invented at Xerox PARC

系统调用 (System Calls)

提供了操作系统提供的有效服务界面

Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use

程序通过高层应用程序接口访问，而不是直接调用系统的调用

常用的 API：

- Win32 API
- POSIX API(包括几乎所有 UNIX、Linux 和 Mac OS X)
- Java API

向系统传递参数的方法：

- 通过寄存器
- 存在内存的块或表中
- 放在或压入堆栈中

系统调用类型

- 进程控制 Process control

    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - wait event, signal event
    - allocate and free memory
- 文件管理 File management
    - create file, delete file
    - open, close
    - read, write, reposition(重定位)
    - get file attributes, set file attributes
- 设备管理 Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices（逻辑连接或断开设备）
- 信息维护 Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
- 通信 Communications
    - create, delete communication connection
    - send, receive message
    - transfer status information
    - attach or detach remote devices

进程控制

- 运行程序需要能正常或非正常地中断其执行 (end 或 abort)
- 当发生错误的时候，就会由内存信息转储 (dump of memory) 并产生一个错误信息
- 装入和执行另一个程序
- 如果新程序终止时控制权返回到现有程序，那么需要保存现有程序的内存映像
- 创建新的作业或线程，需要可以控制他的执行，能决定和重置进程或作业的属性 (优先级、最大允许执行时间)
- 等待作业或进程执行
- 提供转储内存信息的系统调用，帮助调试
- 提供程序的时间表
- 单任务与多任务系统的控制

设备管理

I/O 设备和文件非常相似，在 UNIX 中将这两者合并为文件 - 设备结构

通讯

通讯模型：

- 消息传递模型 (message-passing model)
- 共享内存模型 (shared-memory model)

系统程序

可以分为：

- 文件管理 File manipulation
- 状态信息 Status information
- 文件修改 File modification
- 程序语言支持 Programming language support
- 程序装入和执行 Program loading and execution
- 通讯 Communications
- 应用程序 Application programs

操作系统设计和实现

设计目标

定义系统的目标和规格

系统类型：批处理、 分时、 单用户、 多用户、 分布式、 实时、 通用目标

两个基本类： 用户目标和系统目标

机制（mechanism）和策略（policy）

区分机制和策略对于灵活性很重要

- 机制决定如何做
- 策略决定做什么，策略可能随时间或位置而有所改变。

实现

现代操作系统一般都是用高级语言编写， 如 C 或 C++

代码编写更快， 更为紧凑，容易理解和调试。

缺点仅仅为降低了速度和增加了存储要求

操作系统结构

简单结构

MS-DOS：并没有很好区分接口和功能层次，应用程序可以直接访问基本的 I/O。

UNIX： 由内核和系统程序组成， 物理硬件之上和系统调用接口之下的所有部分都是内核， 这一单一的结构使 UNIX 难以增强

分层方法 Layered

采用自顶向下方法，可先确定总的功能和特征，再划分模块

使用分层法来模块化，最底层 (层 0) 是硬件层，最高层 (层 N) 为用户接口

主要优点： 构造和调试的简单化，每层只能调用较低层的功能和服务。

主要困难：对层的详细定义较难， 效率稍差。

微内核 Microkernel

将非基本部分从内核移走

微内核通常包括最小的进程和内存管理以及通讯功能。

主要功能：使用户程序和运行在用户空间的各种服务之间进行通讯（利用消息传递）

好处：

- 便于扩充操作系统
- 容易移植操作系统到新结构中
- 更加可靠（在内核模式下运行的代码少）
- 更加安全

缺点：由于系统功能总开销的增加而导致系统性能的下降

模块 Modules

模块化的内核：

- 使用面向对象 (object-oriented) 方法
- 每个核心组件分离
- 模块通过接口 (interfaces) 互相调用
- 每个模块都作为内核中的可加载模块

类似于分层，但是更灵活

Solaris 系统：

- 核心内核 core Kernel
    - 调度类 scheduling classes
    - 文件系统 file systems
    - 可加载的系统调用 loadable system calls
    - 可执行格式 executable formats
    - STREAMS 模块 STREAMS modules
    - 杂项模块 miscellaneous modules
    - 设备和总线驱动 device and bug drivers

Mac OS X， 混合结构：

- 程序 ->BSD->Mach
- 程序 ->Mach

虚拟机

提供与基本硬件相同的接口

虚拟用户模式和虚拟内核模式运行在物理用户模式

优点：

- 不同的系统资源具有完全的保护
    - 可以通过共享磁盘或虚拟机网络通讯
- 研究和开发操作系统

实例

- VMware
    - 将 Intel 80x86 硬件抽象为独立的虚拟机
- Java 虚拟机（JVM）
    - 类加载器： 从 Java 程序和 Java API 中加载编译过的. class 文件
    - 执行与平台无关的字节码 Java 解释器
    - JIT(just-in-time)，可以在第一次调用 Java 方法时，转换为本地机器语言，加快运行速度。
- CLR（.Net 的核心，公共语言运行时）

系统生成

System generation， SYSGEN

配置和生成系统

需要确定的信息：

- CPU
- 内存
- 可用设备及其具体调用信息
- 系统选项的参数

生成方法（区别在于大小和通用性，以及硬件变化进行修改的方便性）：

- 系修改源代码，重新编译
- 从预先编译的库中选择模块，将表格连接起来形成操作系统
- 由表驱动，在运行时选择适当的模块（大多数现代操作系统）

系统启动

过程

- 加电
- 运行初始化程序 (或引导程序，在 ROM 中)
    - 初始化 CPU 寄存器，设备控制器到内存
- 引导程序装入操作系统内核到内存
- 操作系统执行第一个进程 init，并等待某些事件发生
- 若事件发生，则进入该事件处理程序

个人计算机中：一个简单的引导程序从磁盘调用一个较复杂的引导程序，再装入内核

CPU 重置：从 ROM 中，指令寄存器被重新加载，并开始执行。（此时 RAM 处于未知状态）

有些系统在 ROM 中保存完整的操作系统

第二部分 进程管理

第 3 章 - 进程

操作系统负责进程和线程管理：

- 用户进程和系统进程的创建和删除
- 进程调度
- 提供进程同步机制
- 进程通信机制
- 进程死锁处理机制

进程（Processes）

正在执行的程序，需要一定的资源来完成任务。是大多数系统的工作单元。

程序本身不是进程，程序只是被动实体，进程是活动实体

一个程序包括：

- program counter
- stack
- data section

内存分配：

- 栈段：编译时设定，由操作系统分配和释放，保存函数参数和返回地址和局部变量等
- 堆区：由程序员分配释放
- 数据段：全局区、静态区和常量区
- 程序代码段：二进制代码

进程状态

- New
- Running
- Waiting / Blocked
    - Blocked Suspend
- Ready
    - Ready Suspend
- Terminated / Exit

每次只有一个进程可以在一个处理器上运行，但多个进程可处于就绪或等待状态。

进程控制块

Process control block， PCB， 也称任务控制块（信息的仓库）

包括：

- 进程状态
- 程序计数器
- CPU 寄存器
- CPU 调度信息
    - 优先级
    - 调度队列的指针
    - 其他调度参数
- 内存管理信息
    - 基址
    - 界限寄存器的值
    - 页表
    - 段表
- 记账信息
    - CPU 时间
    - 实际使用时间
    - 时间界限
    - 记账数据
    - 作业或进程数量
- I/O 状态信息
    - I/O 设备列表
    - 打开的文件列表

CPU 通过保存 PCB 来进行进程间的切换

线程

进程可以一次执行多个线程

进程调度

选择一个可用的进程到 CPU 上执行

调度队列

通常用链表实现。

- 作业队列：Job queue - set of all processes in the system
- 就绪队列：Ready queue - set of all processes residing in main memory, ready and waiting to execute
- 设备队列：Device queue - set of processes waiting for an I/O device
    - 等待特定 I/O 设备的进程列表称为设备队列，每个设备都有自己的设备队列。

操作系统在多个队列之间切换

Linux 中的进程控制块 (PCB) 中是通过结构体表示，包含以下信息

- 进程 id
- 进程的状态

- 调度和内存管理信息
- 打开文件列表
- 指向父进程和所有子进程的指针

我们通常使用队列图表示进程调度：

- 长方形：队列（就绪队列 / 设备队列）
- 圆形：队列服务的资源
- 箭头：系统内进程的流向

当进程分配到 CPU 并执行的时候，可能发生下面几种事件中的一种：

- 进程发出一个 I/O 请求，并放入到 I/O 队列中
- 进程创建一个新的子进程，并等待其结束
- 进程由于中断强制释放 CPU，并被返回就绪队列中

调度程序（Scheduler）

长期调度程序（long-term scheduler）/ 作业调度程序（job scheduler）：从池中选择进程，装入内存准备执行。（创建进程的平均速度必须等于进程离开系统的平均速度，必须合理分配 **I/O** 和 **CPU** 密集类程序）

短期调度程序（short-term scheduler）/ CPU 调度程序：准备执行的进程中选择进程，并为之分配 **CPU**。（通常 100ms 执行一次，比较重视速度）

中期调度程序（medium-term scheduler） 将进程从内存（或 CPU 竞争）中移出，之后可以重新调入内存（进入就绪队列），并从中断处继续执行，称为交换（**swapping**）。

上下文切换（Context switch）

当发生一个中断时，系统需要保存当前运行在 CPU 中进程的上下文，从而在其处理完能恢复上下文。

进程上下文用 PCB 表示，包括：CPU 寄存器的值、进程状态、内存管理信息。

通过执行一个 状态保存 （state save）保存 CPU 当前状态

通过执行一个 状态恢复 （state restore）重新开始运行

硬件支持：有的处理器提供多组寄存器集合，上下文切换只需改变当前寄存器组的指针

进程操作

进程创建

进程在执行过程中，能通过 创建进程系统调用 (Create-process system call) 创建多个新进程。

创建进程成为父进程，新进程成为子进程，形成进程树。

子进程可能从操作系统直接获取资源，也可能只从父进程获取资源。

父进程可能必须在其子进程之间分配资源或共享资源，限制子进程只能使用父进程的资源可以防止创建过多的进程带来的系统超载。

进程创建的初始化数据（或输入）由父进程传递给子进程。

创建新进程：

- 父进程与子进程并发执行
- 父进程等待，直到某个或全部子进程执行完。

新进程的地址空间:

- 子进程是父进程的复制品 (duplicate)
    - 具有相同的程序和数据
- 子进程装入另一个新程序

UNIX 中

通过 `fork()` 创建新进程。新进程通过复制原来进程的地址空间而成。

对于新进程，系统调用 `fork()` 返回值为 0，对于父进程，返回值为子进程的进程标识符。

在 `fork()` 之后，一个进程使用 `exec()` ，可以用新程序来取代进程的内存空间。

这样两个进程就可以互相通信并可以通过各自的方式执行。

如果子进程运行时没有什么可以做，那么就可以采用 `wait()` 把自己移出就绪队列来等待子进程的终止。

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void main(int argc, char *argv[]) {
    pid_t pid;
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        exit(-1);
    } else if (pid == 0) {
        execlp("/bin/ls", "ls", NULL);
    } else {
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

PS: `Copy on write` : 需要写的时候才发生内存复制。

Windows 中

使用 `Win32 API` 中的 `CreateProcess()` 函数， 执行时，函数将一个特殊程序装入子进程的地址空间。需要至少 10 个参数。

`CreateProcess()` 需要两个参数 `STARTUPINFO` 和 `PROCESS_INFORMATION`

`STARTUPINFO` 指明新进程的许多特性，如窗口大小、标准输入及输出文件句柄

`PROCESS_INFORMATION` 包括一个句柄以及新生成进程和线程的标识

调用前，需要使用 `ZeroMenory()` 为这两个结构清空内存

`Win32 API` 相对于 `Linux` 的 `wait()` 是 `WaitForSingleObject()`

进程终止

当进程执行完调用 `exit()` 后，进程终止，返回状态值到父进程，所有进程资源被释放。

父进程通过标识符使用适当的系统调用也可以终止子进程。

对于某些系统当父进程终止的时候，子进程也会被终止，而某些系统父进程终止后，子进程会以 `init` 进程作为父进程。

进程间通信

并发执行的进程可以分为独立进程和协作进程

独立进程： 不能影响其他进程或被其他进程所影响， 不与任何其他进程共享数据

协作进程： 能影响其他进程或被其他进程所影响， 与任何其他进程共享数据

提供环境允许进程协作的优点:

- 信息共享 Information sharing
- 提高运算速度 Computation seed-up： 把特定任务分成子任务并发执行
- 模块化 Modularity： 将系统功能分成独立的进程或线程
- 方便 Convenience： 单用户也可以同时执行多任务

协作进程需要一种进程间通信机制（interprocess communication, IPC）来允许进程相互交换数据与信息。

两种基本通信模式:

- 共享内存
    - 更快的速度
- 信息传递
    - 更方便
    - 交换少量数据

共享内存系统

生产者进程产生信息以供消费者进程消费，共享内存时解决生产者 - 消费者问题方法中的一种。

有两种缓冲方式：

- 无限缓冲（unbounded-buffer）：消费者可能不得不等待新的项， 生产者总是可以产生新项。
- 有限缓冲（bounded-buffer）：缓冲大小固定，如果缓冲为空，消费者等待，如果缓冲为满，生产者等待。

生产者进程：

```
item nextProduced

while (true) {
        where(((in + 1) % BUFFER_SIZE) == out); // 当缓冲满了的时候必须等待
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

消费者进程:

```
item nextConsumed
while (true) {
        while (in == out); // 当缓冲为空的时候必须等待
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

操作系统服务

消息传递系统 IPC

常用于分布式环境中，如 Web

消息传递提供一种机制允许进程不必通过共享地址空间来实现通讯与同步

通信需要有通信线路（communication link）

通讯线路的实现：

- 物理：共享内存、硬件总线
- 逻辑：逻辑属性 properties

逻辑实现线路和发送接收操作的方法：

- 直接或间接通信
- 同步或异步通信
- 自动或显式缓冲

命名

对于直接通信：

对称寻址：

- `send(P, message)` ：发送信息到进程 P
- `receive(Q, message)` ：接受来自进程 Q 的信息

属性：

- 每对需要通信的进程之间自动建立线路，进程仅需知道相互通信的标识符

- 一个线路只与两个进程相关
- 每对进程之间只有一个线路

非对称寻址：

- `send(P, message)`：发送信息到进程 P
- `receive(id, message)`：接受来自任何进程的信息，id 设置成与其通信的进程名称

这两种寻址方式的缺点在于限制了进程定义的模式化，改变进程名字需要检查其他所有进程的定义。

对于间接通信：

通过邮箱和端口来发送或接受信息。

一个进程可能通过许多不同的邮箱与其他进程通信，但两个进程仅在其共享至少一个邮箱时可以相互通信。

- `send(A, message)`： 发送一个消息到邮箱 A
- `receive(A, message)`：接受来自邮箱 A 的消息

对于这种方案，通信线路有如下属性：

- 只有在两个进程共享一个邮箱，才能建立通信线路
- 一个线路可以与两个或更多的进程相关联
- 两个通信进程之间可有多个不同的线路，每个线路对于一个邮箱
- 连接可能是单向 `unidirectional` 或者双向 `bi-directional`

操作系统需提供的机制：

- 创建新邮箱
- 通过邮箱发送和接受消息
- 删除邮箱

邮箱的拥有权和接受权可以通过系统调用传递给其他进程。

同步

消息传递可以时阻塞或非阻塞（同步或异步）

- 阻塞 send： 发送进程阻塞， 直到消息被接受进程或邮箱所接受
- 非阻塞 send： 发送进程发送消息并再继续操作
- 阻塞 receive： 接收者阻塞， 直到有消息可用
- 非阻塞 receive： 接收者收到一个有效信息或空信息

当 `send()` 和 `receive()` 都阻塞， 发送者和接收者之间就有一个 集合点（`rendezvous`）

缓冲

- 无缓冲
    - 零容量: 阻塞
- 自动缓冲
    - 有限容量： 线路满后阻塞

- 无限容量： 不阻塞

客户机 - 服务器系统通信

除了共享内存和消息传递，还有三种通信方法： Socket、远程过程调用（RPC）和 Java 的远程方法调用（RMI）

Socket

Socket（套接字）可定义为通信的端点。一对通过网络通信的进程需要使用一对 Socket（即每个进程各有一个）。

Socket 由 IP 地址和一个端口号连接组成

服务器通过 监听制定端口 来等待进来的客户请求， 一旦受到请求， 就接受来自客户 Socket 的连接， 从而完成连接。

Java 提供了三种不同类型的 Socket

- 面向连接（TCP）Socket： Socket 类
- 无连接（UDP）Socket： DatagramSocket 类
- 多点传送 Socket： DatagramSocket 的子类（允许数据发送到多个接收者）

使用 Socket 通信，虽然常用和高效， 但是属于较为低级的分布式进程通信。原因之一在于 Socket 只允许在通信进程之间交换无结构的字节流

远程过程调用

Remote Procedure Calls, RPC

RPC 抽象层远程调用机制，用于通过网络连接系统

存根 Stubs： 用于服务器上的实际程序的客户端代理

客户端使用存根定位服务器，并提供参数

服务器接收信息，解析后运行指定的程序

第 4 章 线程

概述

线程是 CPU 使用的基本单元。它与属于同一进程的其他线程共享代码段、数据段和其他操作系统资源。

组成：

- 线程 ID
- 程序计数器
- 寄存器集合
- 栈

优点

- 响应度高：即使部分阻塞，程序也能继续执行，从而增加了对用户的相应速度
- 资源共享：线程默认共享他们所属的进程的内存和资源
- 经济：进程创建所需要的内存和资源的分配比较昂贵，创建和切换线程比较经济

- 多处理器体系结构的利用：多线程可以并行在不同的 CPU 上

多线程模型

有两种不同方法提供线程支持：

- 用户层的用户线程
- 内核层的内核线程（由操作系统直接支持和管理）

多对一模型

将许多用户级线程映射到一个内核线程。

线程管理是由线程库在用户空间进行的，因而效率比较高

一对一模型

将每个用户线程映射到一个内核线程

开销比较大

多对多模型

开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。

并且在一个线程执行阻塞系统调用的时候，内核能调度另一个线程来执行。

二级模型

类似于多对多模型，二级模型还允许用户线程绑定在一个内核线程里面

线程库

为程序员提供创建和管理线程的 API

两种方法:

- 在用户空间提供一个没有内核支持的库
- 执行一个由操作系统直接支持额内核级的库

多线程问题

系统调用 fork() 和 exec()：

fork:

- 可以复制所有线程
- 仅仅复制调用 fork 系统调用的线程

exec:

如果一个线程调用 exec()，那么整个进程都会被替换

如果 fork 后立即调用 exec，那么就没有必要复制所有线程

线程取消 Thread Cancellation

在线程执行完成后中止

取消方法：

- 异步取消 (asynchronous cancellation)：立即终止目标线程
- 延迟取消 (deferred cancellation)：允许目标线程周期性地检查是否应该取消该线程，这些检查点称为取消点 (cancellation point)

信号处理 Signal Handling

信号处理程序用于通知进程某个特性事件发生了

可以为同步或异步接收

模式：

- 信号是由特定事件产生的
- 生成的信号传递到进程
- 一旦发送，信号必须加以处理

信号的处理：

- 默认的信号处理程序 (default signal handler)
- 用户定义的信号处理程序

信号发送的选择（依赖于信号的类型）：

- 将信号传递到信号所应用的线程
- 将信号传递给进程中的每个线程
- 将信号传递给进程中的某些固定线程
- 指定一个特定的线程来接收进程的所有信号

线程池

优点：

- 通常比创建新的线程要快
- 限制了任何时候的可用线程数

线程的特定数据 Thread Specific Data

允许每个线程有自己的一定的数据副本

当使用线程池的时候有用（不是自己创建进程的时候）

调度程序激活

Scheduler Activations

多对多和二级模型都需要内核与线程库之间的通信（允许动态调整内核线程的数量以保证最好的性能）

内核通过 upcall 与线程库通信，允许应用程序维护正确的内核线程数

内核提供一组虚拟处理器 (LWP) 给应用程序，应用程序可调度用户线程到一个可用的 LWP 中。

upcall 处理句柄必须在虚拟处理器上运行。

第 5 章 CPU 调度

基本概念

当一个程序必须等待时，操作系统会从该进程拿走 CPU 的使用权，而将 CPU 交给其他进程

CPU-I/O 区间

进程执行由 CPU 执行和 I/O 等待周期组成，进程在这两个状态之间切换。

CPU 约束程序可能有少量的长 CPU 区间

I/O 约束程序通常具有很多短 CPU 区间

CPU 调度程序

当 CPU 空闲，操作系统从就绪队列选择一个进程执行，从内存中选择并为之分配 CPU

就绪队列的实现可以为：FIFO 队列， 优先队列，树或简单的无序链表，其内容通常为进程控制块 (PCB)

抢占调度

CPU 调度决策的发生：

- 运行状态切换到等待状态
- 运行状态切换到就绪状态
- 等待状态切换到就绪状态
- 进程终止

第一种和第四种没有选择只有调度， 称为非抢占的 (non-preemptive) 或协作的(cooperative)

否则调度方案就是抢占的 (preemptive)

抢占调度需要特别的硬件（如计时器）的支持，因此有的设备只能使用协作调度

注意: 抢占调度对访问共享数据有代价，对操作系统内核设计也有影响。

分派程序 Dispatcher

用于将 CPU 的控制交给由短期调度程序选择的进程，功能:

- 切换上下文
- 切换到用户模式
- 跳转到用户程序的合适位置，以重新启动程序

分派程序停止一个进程而启动另一个所要花的时间称为分派延迟 (dispatch latency)

调度准则 Scheduling Criteria

- CPU 使用率 CPU utilization： 使得 CPU 尽可能忙
- 吞吐量 Throughput： 指一个时间单元内所完成进程的数量

- 周转时间 Turnaround time: 从进程提交到进程完成的时间段。包括等待进入内存，在就绪队列中等待， 在 CPU 上执行和 I/O 执行
- 等待时间 Waiting time ： 在就绪队列中等待所花的时间之和
- 响应时间 Response time： 从提交请求到产生第一响应的时间

需要使得 CPU 使用率，吞吐量最大化，使周转时间，等待时间，响应时间最小化。

调度算法

先到先服务调度

first-come, first-served(**FCFS**) Scheduling algorithm

先请求 CPU 的进程先分配到 CPU

使用 FIFO 队列实现

通常平均等待时间比较长。

护航效果 **(convoy effect)**: 所有其他进程都在等待一个大进程释放 CPU，导致 CPU 和设备的利用率变得更低

FCFS 调度算法是非抢占的。

最短作业优先调度

shortest-job-first (**SJF**) scheduling algorithm

优先赋给具有最短 CPU 区间的进程

SJF 调度算法可证明为最佳的

难点在于如何知道下一个 CPU 区间的长度，因此一般用于长期调度，将用户提交作业时所指定的进程时间极限作为长度。

预测算法：

$$T_{(n+1)} = \alpha t_n + (1-\alpha)T_n$$

当 $\alpha = 0$ 最近的历史没有统计

当 $\alpha = 1$ 只是根据上一次的运行时间

对于短期调度，通常去根据以前的 CPU 区间长度来预测下一个。预测值为之前的指数平均

SJF 调度算法可以为抢占或者时非抢占的。

抢占式的 SJF 可抢占当前运行的进程，有时称为最短剩余时间优先调度 (shortest-remaining-time-first scheduling, SRTF)

非抢占式的允许当前运行的进程先完成其 CPU 区间。

优先级调度

SJF 算法可以作为通用优先级调度算法 (priority scheduling algorithm) 的一个特例，优先级为下一个 CPU 区间的倒数

每个进程都有一个优先级与其相关联，具有最高优先级的进程会分配到 CPU，具有相同优先级的按 FCFS 顺序调度。

优先级可以通过内部或者外部定义，内部通常使用一些测量数据，如：时间极限，内存要求，打开文件的数量和平均 I/O 区间和平均 CPU 区间。

优先级调度可以式抢占或者式非抢占的。

主要问题：无穷阻塞 (indefinite blocking) 或饥饿(starvation)

可以运行但缺乏 CPU 的进程被认为是阻塞的，他在等待 CPU，调度算法会使某个低优先级进程无穷等待 CPU

解决方案：老化 (aging)， 逐渐增加在系统中等待很长时间的进程的优先级。

轮转法调度

round-robin, RR

专门为分时系统设计，在 FCFS 上增加了抢占以切换进程。改进了响应时间。

定义一个较小的时间单元，称为时间片 (time quantum. or time slice)，通常为 10~100ms，将就绪队列作为循环队列，为每个进程分配不超过一个时间片的 CPU

RR 策略的平均等待时间通常较长，性能很大程序依赖于时间片的大小，如果时间片很小，那么 RR 算法就称为处理器共享

性能：

当时间段比较长的时候：退化为 FIFO

当时间段比较短的时候：时间段必须大于上下文切换的时间，否则开销会很大

多级队列调度

multilevel queue scheduling

将就绪队列分成多个独立队列。根据进程的属性（内存大小，进程优先级， 进程类型）

每个队列有自己的调度算法，如

- 前台队列：RR 算法
- 后台队列：FCFS 算法

队列之间也必须有调度，通常采用固定优先级抢占队列，比如前台队列对于后台队列有绝对的优先级，不过固定的优先级可能会引起饥饿

队列之间也可以划分时间片，每个队列有一定的 CPU 时间（如，80% 的时间给前台队列，20% 给后台）

多级反馈队列调度

multilevel feedback queue scheduling

比多级队列调度更加灵活，但是开销也更大

允许进程在队列之间移动，主要思想是根据不同的 CPU 区间的特点以区分进程。如果进程使用过多 CPU 时间，那么它就会被转移到更低优先级的队列，如果低优先级队列中等待时间过长，那么也会被转移到更高优先级队列。这种老化防止了饥饿的发生。

通常，可以由以下参数来定义

- 队列数量
- 每个队列的调度算法
- 用以确定何时升级到更高优先级队列的方法
- 用以确定何时降级到更低优先级队列的方法

- 用以确定进程在需要服务时应进入哪个队列的方法

多级反馈队列调度程序为最通用的 CPU 调度算法，也是最复杂的。

如：

首先进程进入第一级 FCFS 队列，如果 8ms 内不能完成，就移到第二级 FCFS 队列，如果 16ms 内不能完成，就进入三级 FCFS 队列

高级队列的优先级最高，仅当高级队列为空，才调度较低优先级的队列。如果低优先级的进程被抢占，则放入原队列的末尾。

如果进程请求 I/O，让出了 CPU 则 I/O 完成后，放入比原来队列高一级的队列中

多处理器调度

多个 CPU 可实现负载分配 (load sharing)，调度问题也更加地复杂

即使对于同构多处理器，也有一些调度限制，比如一个 I/O 设备与一个处理器通过私有总线相连，那么必须把使用改设备的进程调度到这个 CPU 上去运行。

实时调度：

- Hard real-time systems：保证在一定时间内完成重要的任务
- Soft real-time systems：重要进程的优先级比其他的高

多处理器调度的方法

- 非对称多处理 (asymmetric multiprocessing)
    - 一个处理器（主服务器）处理所有的调度决定、I/O 处理以及其他系统活动
    - 其他处理器只执行用户代码
    - 更加简单，减轻了对数据共享的需要
- 对称多处理 (symmetric multiprocessing, SMP)
    - 每个处理器自我调度
    - 调度通过每个处理器检查共同就绪队列并选择一个进程来执行
    - 必须保证两个处理器不能选择同一个进程，并且进程不会从对联中丢失

处理器亲和性

当一个特定的进程在处理器上运行的时候，进程最近访问的数据进入处理器缓存，缓存的失效和重新构建代价时很高的。

处理器亲和性： 避免将进程从一个处理器移到另一个处理器，使得一个进程在同一个处理器上运行

- 软亲和性 soft affinity： 进程可能在处理器之间移动
- 硬亲和性 hard affinity：进程被指定不允许移到其他处理器

负载平衡

设法将工作负载平均地分配到 SMP 系统的所有处理器上。

只是对于那些拥有子集私有的可执行进程的处理器是必要的，对于有共同队列的系统是不需要的。

通常有两种方法：

- push migration

    - 一个特定的任务周期性地检查每个处理器上的负载，发现不平衡就将进程从超载 CPU 移动到空闲 CPU
- pull migration
    - 空闲处理器从一个超载处理器 pull 一个等待任务

通常这两种技术并行地实现

负载平衡通常会抵消处理器亲和性的优点。

对称多线程

SMT

SMP 是基于多个物理处理器，SMT 基于多个逻辑处理器，也称为超线程技术 (Hyperthreading)

SMT 的思想是在同一个物理处理器生成多个逻辑处理器，向操作系统呈现一个多逻辑处理器的视图。每个逻辑处理器都有自己的架构状态

SMT 是硬件提供的。

线程调度

线程：

- 用户线程
- 内核线程

竞争范围

进程竞争范围（process-contention scope, PCS）（Local Scheduling）：线程库调度用户级线程到一个有效的 LWP 上运行（根据优先级）

系统竞争范围（system-contention scope, SCS）（Global Scheduling）：调度内核线程到 CPU

算法评估

定义准则：

- 最大化 CPU 使用率，同时要求最大相应时间为 1s
- 最大化吞吐量（平均周转时间与总地执行时间成正比）

确定类型

评估方法:

- 分析评估法 (analytic evaluation)：使用给定的算法和系统负荷，产生一个公式或数字来评估
- 确定模型法 (deterministic modeling)：采用特殊预先确定的符合，计算在给定负荷下算法的性能

排队模型

排队网络分析 (queueing-network analysis)：知道到达率和服务率，计算使用率，平均队列长度，平均等待时间等

Little 公式： $n = \lambda \times W$

第 6 章 进程同步

进程或线程异步执行并且共享数据，那么就有可能不能正确运行

竞争条件 race condition： 多个进程并发访问和操作同一数据并且执行结果与访问发生的特定顺序有关

为了避免竞争条件，需要确保一段时间内只有一个进程能操作变量，要求进行一定形式的进程同步

因此需要进程同步 (process synchronization) 和协调 (coordination)

临界区问题

临界区 (critical section)： 每个进程中的一个代码段，在该区内进程可能改变共同变量等操作。

每个进程必须请求允许进程临界区，请求进入的代码称为进入区 (entry section), 临界区后可有退出区 (exit section), 其他代码为剩余区 (remainder section)

临界区问题的三个要求：

- 互斥 mutual exclusion: 当一个进程进入临界区，那么其他进程不能再临界区内运行。(无空等待)
- 前进 progress: 如果没有进程再其临界区内执行且有进程需要进入临界区，那么只有哪些不在剩余区内执行的进程可以参加选择，确定谁能进入临界区，而且这种选择不能无限推迟（有空让进）
- 有限等待 bounded waiting: 从一个进程做出进入临界区的请求，直到道该请求允许为止，其他进程允许进入临界区的次数有限。

处理临界问题

抢占内核 preemptive kernel， 需要设计以确保内核数据结构不会导致竞争条件。更适合实时编程，允许实时进程抢占内核默认运行的其他类型，响应更快。

非抢占内核 nonpreemptive kernel， 从内核数据结构上就不会导致竞争条件

让权等待：当进程不能进入自己的临界区的时候，释放处理器，以免进程进入盲等

一些解法

双标志、先检查：可能两个都进入临界区

双标志、后检查：可能两个都不能进入临界区

Peterson 算法

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    临界区
    flag[i] = false;
    剩余区
} while (true);
```

适用于两个进程再临界区与剩余区间交替执行

此算法：

- 互斥成立
- 前进要求满足
- 有限等待要求满足

硬件同步

很多系统提供临界区的硬件支持

单处理器：关中断，确保当前指令序列执行不被中断，常为非抢占内核使用

多处理器：提供特殊硬件指令允许原子地检查和修改字的内容或交换两个字的内容

```
do {
    请求锁
        临界区
    释放锁
        剩余区
} while(true);
```

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}

do {
    while (TestAndSet(&lock));
    // do something
    lock = false;
} while (true);

void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

do {
    key = true;
    while (key == true) Swap(&lock, &key);
    // do something
    lock = false;
} while (true);
```

以上为两种实现方法

优点：

- 适用于任意数目的进程，在单处理器和多处理器
- 简单
- 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量

缺点：

- 等待需要耗费 CPU 时间，不能实现让权等待
- 可能饥饿，优先级问题
- 可能死锁， 优先级问题，如果有一个更高优先级的进程在临界区执行时中断并且需要这个进程的资源，就会造成死锁

信号量

semaphore

一个同步工具

信号量 S 是个整型变量，除初始化外，只能通过两个标准原子操作 `wait()` 和 `signal()` 访问，也称为 P 和 V

```
wait(S) {
    while(S <= 0);
    S--;
}
signal(S) {
    S++;
}
```

使用方法：

```
mutex = 1;
do {
    wait(mutex);
    // do something
    signal(mutex)
} while (true);
```

缺点：忙等待。通过循环浪费了 CPU，称为自旋锁 (spinlock)

改进： 变成阻塞自己，将进程放入与信号量相关的等待队列， 并切换进程状态为等待状态。

```
typedef struct {
        int value;
    struct process *list;
} semaphore;
```

当一个进程必须等待信号量，那么就加入进程链表， `signal()` 会从等待链表中取一个进程以唤醒。

```
wait (semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block(); // 挂起调用它的进程
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P); // 重新启动阻塞进程P的执行
    }
}
```

`value` 就是等待的进程数。

大于等于 0 的时候就是可用资源数。

小于 0 就是等待进入临界区的进程个数

如果定义信号量 flag 为 0，可以先 `signal` 再 `wait` ， 确认操作完成后再继续执行。

对于单处理器，可以执行 `wait` 和 `signal` 操作时简单地禁止中断，对于多处理器，必须禁止每个处理器的中断，或者提供其他加锁技术（如自旋锁 Spinlock，特点就是忙等待），使得这两个命令可以原子地执行。

并没有完全取消忙等，而是取消了应用程序进入临界区的忙等。

`wait` 和 `signal` 必须成对出现:

- 互斥操作时，处于同一进程
- 同步操作时，不在同一进程

如果两个 `wait` 相邻，那么他们顺序很重要（同步操作应该在互斥操作之前）， `signal` 则不重要。

优点：简单，表达能力强，能解决任何同步互斥问题

缺点：不够安全，当使用不当会出现死锁，实现复杂

死锁和饥饿

当信号量使用顺序不对，就有可能造成死锁。

当两个或多个进程无限等待一个事件，而该事件只能由这些等待进程之一来产生，那么这些进程就会称为死锁 (deadlocked)

当进程再信号量内无限期等待，就会导致无限期阻塞 (indefinite blocking) 或饥饿 (starvation)

经典同步问题

有限缓冲问题

- mutex 提供对缓冲池访问的互斥要求, 初始化为 1
- empty 表示空缓冲项，初始化为 n
- full 表示满缓冲项，初始化为 0

```
// 生产者
do {
    // produce an item in nextp
    wait(empty);
    wait(mutex);
    // add nextp to buffer
    signal(mutex);
    signal(full);
} while (true);

// 消费者
do {
    wait(full);
    wait(mutex);
    // r3emove an item from buffer to nextc
    signal(mutex);
    signal(empty);
    // consume the item in nextc
} while (true);
```

读者 - 写者问题

- 第一读者 - 写者问题（读者优先）： 要求没有读者需要保持等待除非已有一个写者已获得允许以使用共享数据库，即没有读者会因为有一个写者在等待而会等待其他读者的完成. (即使有一个写者在等待，读者也会无视读者的锁而进行并发地读取，可能导致写者饥饿)
- 第二读者 - 写者问题（写者优先）： 一旦写者就绪，那么写着就会尽可能快地执行写操作，即如果是个写者等待访问对象，那么不会有新读者开始读操作（一旦有写者等待，读者就会不会开始，可能导致读者饥饿

第一读者写者问题解决

```
semaphore mutex = 1, wrt = 1;
// mutex 修改readcount的互斥
// wrt 写者的互斥
int readcount = 0;

// 写者进程结构
do {
    wait(wrt);
    ...
    // writing is performed
    ...
    signal(wrt);
```

```
    signal(wrt);
} while (true);

// 读者进程结构
do {
    wait(mutex);
    readcount++;
    if (readcount == 1) { // 当前没有读者，需要等待/获取写者互斥
        wait(wrt);
    }
    signal(mutex);
    ...
    // reading is performed
    ...
    wait(mutex);
    readcount--;
    if (readcount == 0) { // 所有读者退出，释放写者互斥
        signal(wrt);
    }
    signal(mutex);
} while (true);
```

第二读者写者问题

```
semaphore rmutex = 1, wmutex = 1, readTry = 1, resource = 1;
// rmutex 修改readcount的互斥
// wmutex 修改writecount的互斥
// readTry 锁住进入读者队列
// resouce 写者修改资源的互斥
int readcount = 0, writecount = 0;

// 写者进程结构
do {
    wait(wmutex);
    writecount++;
    if (writecount == 1) { // 当前没有写者，需要先锁住读者队列
        wait(readTry);

    }
    signal(wmutex);
    wait(resouce); // 需要获得资源的独占访问
    ...
    // writing is performed
    ...
    signal(resouce);
    wait(wmutex);
    writecount--;
```

```
23        if (writecount == 0) { // 所有写者退出，释放读者队列的锁
24            signal(readTry);
25        }
26        signal(wmutex);
27
28    } while (true);
29
30    // 读者进程结构
31    do {
32        wait(readTry); // 获取进入读者队列的锁
33        wait(rmutex);
34        readcount++;
35        if (readcount == 1) { // 当前没有读者，需要等待/获取资源独占锁
36            wait(resource);
37        }
38        signal(rmutex);
39        signal(readTry);
40        ...
41        // reading is performed
42        ...
43        wait(rmutex);
44        readcount--;
45        if (readcount == 0) { // 所有读者退出，释放写者互斥
46            signal(resource);
47        }
48        signal(rmutex);
49    } while (true);
```

某些系统提供读写锁

多个进程可允许并发获取读模式的读写锁，只有一个进程可以为写操作获取读写锁

需要读写锁的情况：

- 当可以区分哪些进程需要读共享数据哪些进程只需写共享数据、
- 当读者进程数比写者进程多时。

哲学家进餐问题

典型的同步问题，并发控制的问题。

需要在多个进程之间分配多个资源且不会出现死锁和饥饿

一个简单的解决方法就是每一个资源使用一个信号量表示

```
1    emaphore chopstick[5]; // 初始化为1
2    do {
3        wait(chopstick[i]);
4        wait(chopstick[(i + 1) % 5]);
```

```
5       // eat
6       signal(chopstick[i]);
7       signal(chopstick[(i + 1) % 5]);
8       // think
9   } while (true);
```

但是这个方法会导致死锁

我们需要保证：

- 最多允许 4 个哲学家同时坐在桌子上
- 只有两只筷子都可用的时候才允许一个哲学家拿起他们（必须在临界区内拿起）
- 使用非对称解决方法
    - 奇数哲学家先拿起左边的筷子，然后拿起右边的筷子，而偶数哲学家则相反

但是还是不能解决饥饿的问题

管程 Monitors

使用信号量还是可能导致错误，或者调用顺序一旦不被遵守，那么就会出现死锁，因此需要管程, monitor

使用

管程类型提供了一组由程序员定义的、在管程内互斥的操作。

管程类型的表示不能直接为各个进程所使用，在管程内定义的子程序只能访问位于管程内那些局部声明和变量形式参数，局部变量只能被局部子程序访问。

管程结构确保一次只有一个进程 能在管程内活动， 但是还需一些额外的同步机制，由条件 (condition) 结构来提供。

```
1   condition x, y;
2   x.wait(); // 调用操作的进程会被挂起
3   x.signal(); // 重新启动一个悬挂的进程，如果没有，那么就没有任何操作（与信号量不同）
```

管程的语法

```
1   monitor monitor name {
2       // shared variable declarations
3       procedure P1 (...) {
4           ...
5       }
6       procedure P2 (...) {
7           ...
8       }
9       ...
10      procedure Pn (...) {
11          ...
12      }
```

```
    initalization code (...) {
        ...
    }
}
```

当操作 x.signal() 被进程 P 调用时， Q 被唤醒， 进程 Q 等待直到 P 离开管程，或者等待另一个条件，然后再执行。

在并行 Pascal 中，当进程 P 执行操作 signal() 时，它会立即离开管程，因此进程 Q 会立刻重新执行。

哲学家进餐问题的管程解决方案

需要区分哲学家三种状态，引入数据结构

```
enum {THINKING, HUNGRY, EATING} state[5];
```

哲学家只有再其两个邻居不在进餐的时候才能将 state[i] 设置为 eating

还需要声明 condition self[5];

其中哲学家 i 在饥饿且又不能拿到需要筷子时可以延迟自己。

```
monitor db {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) {
            self[i].wait();
        }
    }

    void putdown(int i) {
                state[i] = THINGKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
           (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {

            state[i] = EATING;
            self[i].signal();
        }
    }

```

```
    initialization_code() {
        for (int i = 0; i < 5; i++) {
            state[i] = THINKING;
        }
    }
}

// 当哲学家i需要进餐
dp.pickup(i);
eat();
dp.putdown(i);
```

这样就可以保证相邻两个哲学家不会同时进餐，并且在哲学家进餐完毕会唤醒左右两个哲学家（如果他们正在等待并且已经可以开始吃），这样就不会出现死锁，但是还是有可能饿死。

基于信号量的管程实现

对于每个管程，都有一个信号量 `mutex = 1`， 进程在进入管程之前必须执行 `wait(mutex)`， 离开管程之和必须执行 `signal(mutex)`

因为信号进程必须等待，直到重新启动的进程离开或等待，因此引入了信号量 `next = 0` 以供信号进程挂起自己，然后提供一个整数变量 `next_count` 对挂起的进程数量进行计数

```
wait(mutex); // 保证这个管程只有一个在运行
...
if (next_count > 0) {
        signal(next); // 唤醒下一个
} else {
    signal(mutex);
}
```

实现条件变量（引入信号量 `x_sem` 和整数变量 `x_count`）：

条件变量的两个操作：

- `x.wait()`： 调用操作的进程被挂起，调用管程的其他程序（有的话）
- `x.signal()`：恢复调用 `x.wait()` 的进程

```
// x.wait()
x_count++;
if (next_count > 0) {
    signal(next);
} else {
    signal(mutex);
}
wait(x_sem);
```

```
 9   x_count--;
10   ... // 进行操作
11   // signal(next); ???
12
13   // x.signal()
14   if (x_count > 0) {
15       next_count++;
16       signal(x_sem);
17       wait(next);
18       next_count--;
19   }
```

管程内的进程重启

简单的解决方案时使用 FCFS 顺序，但是很多情况下可以使用条件等待构造 `x.wait(c)`

c 为整数，称为优先级， 会与悬挂进程的名称一起存储，当执行 `x.signal()` 最小优先级的程序会被启动。

管程不能正常运行的情况：

- 一个进程在没有先获得资源访问权限就访问资源
- 一个进程在获得资源访问权限之后就不释放资源
- 一个进程视图释放一个从来没有请求的资源
- 一个进程可能请求同一资源两次（中间没有释放资源）

同步实例

Pthread 同步

Pthread API 为线程同步提供：

- 互斥锁（Pthread 的基本同步技术，保护临界区代码）
- 条件变量

不可移植的扩展：

- 自旋锁
- 读写锁

原子事务

这一部分在数据库管理原理与设计系统的事务部分有详细介绍

第 7 章 死锁

系统模型

系统拥有一定数量的资源，分布在若干竞争进程之间。资源可以分成多种类型，每种类型有一定数量的实例。

同类型的资源：一个进程申请某个资源类型的一个实例，同类的任何实例都可以满足要求，那么就是相同的。

进程使用资源的流程:

- 申请：如果不被允许，那么就必须等待，直到允许
- 使用
- 释放

当一组进程中的每个进程都在等待一个事件，而这个事件只能由这一组进程的另一进程引起，那么这组进程就处于死锁状态

死锁特征

必要条件

- 互斥 Mutual exclusion： 至少有一个资源处于非共享模式。
- 占有并等待 Hold and wait：一个进程必须占有至少一个资源，并等待另一资源（被其他进程占有）
- 非抢占 No preemption： 资源不能被抢占，只能等待自动释放
- 循环等待 Circular wait：一组等待进程 $P_0, P_1, .., P_n$ 一个进程的等待资源被下一个进程占用

四个条件必须同时满足才会出现死锁。循环等待意味者占有并等，这四个条件并不是完全独立。

资源分配图

Resource-Allocation Graph

节点类型：

- 系统活动进程的集合（圆形）
- 系统所有资源类型的集合（一个矩形表示一类资源，里面的点表示资源）

边：

- 申请边：进程指向资源
- 分配边：资源指向进程

如果分配图没有环，那么系统就没有进程死锁。如果有，那么就可能存在死锁

如果每个资源类型刚好有一个实例，那么有环就意味着出现了死锁。

死锁处理方法

- 使用协议预防或避免死锁
- 允许系统进入死锁状态，然后检测它，并加以恢复
- 忽视这个问题，认为死锁不可能发生

然而绝大多数系统使用第三种方法，这就需要我们自己处理死锁。

死锁预防， deadlock prevention： 确保至少一个必要条件不成立，通过限制如果申请资源的方法来预防

死锁避免， deadlock avoidance： 操作系统事先得到有关进程申请资源和使用资源的额外信息，然后确定对于一个申请是否应该等待。

死锁预防

只要确保至少一个必要条件不成立。

互斥

使用共享资源，比如只读文件，不过有些资源本身就是非共享的。

占有并等待

保证当一个进程申请一个资源时，他不能占有其他资源。

- 在每个进程执行前申请并获得所有的资源。
- 运行进程在没有资源时才能申请资源，也就是在申请更多资源时候必须释放当前资源

缺点：

- 资源利用率 (resource utilization) 可能比较低
- 可能发生饥饿

非抢占

使用协议：如果一个进程占有资源并申请另一个不能立即分配的资源，那么其现已分配的资源都可以被抢占。也就是说这些资源被隐式释放了。当进程获得其原有资源和所申请的新资源时，才能重新执行。

适用于状态可以保存和恢复的资源，如 CPU 寄存器，内存，但不适用于打印机和磁带驱动器。

循环等待

对所有资源类型进行完全排序，并按递增顺序申请资源。

使用协议：

> 每个进程只按递增顺序申请资源，如果需要同一资源类型的多个实例，那么必须同时申请。
>
> 当一个进程需要前面的资源时，必须释放后面的资源。

这样一来，循环等待就不可能成立。

我们可以使用锁顺序验证器，检测是否可能出现死锁

死锁避免

使用死锁预防会带来低设备使用率和系统吞吐率

因此我们可以获得以后如何申请资源的附加信息，然后决定进程是否因申请而等待。

每次申请要求系统考虑现有可用资源，现已分配给每个进程的资源和每个进程将来申请与释放的资源，以决定当前申请是否满足或必须等待，从而避免死锁发生的可能性。

最简单的模型：要求每个进程说明他们可能需要的每种资源实例的最大需求，构造一个算法确保系统绝不会进入死锁状态

死锁避免 (deadlock-avoidance) 算法动态地检测资源分配状态以确保循环等待条件不可能成立。

资源分配状态由以下数据决定：

- 可用资源
- 已分配资源
- 进程最大需求

安全状态

安全：系统能按某个顺序为每个进程分配资源（不超过其最大值）并能避免死锁

也就是：存在一个安全序列

安全序列：

对于每个进程，进程仍然可以申请的资源数小于当前可用资源加上序列前面的资源所占用的资源。

不安全状态可能导致死锁

避免算法的思想：简单地确保系统始终处于安全状态

资源分配图算法

引入一个需求边：表示某个进程在将来每个时刻申请某个资源（用虚线表示）

当进程申请资源的时候，需求边就变成申请边

当进程释放资源的时候，申请边就变成需求边

只有当需求边变成申请边的时候没有环，才允许申请。

不适用于每种资源由多个实例的资源分配系统

银行家算法

对于每种资源有多个实例的资源分配系统。

当新进程进入系统，必须说明其可能需要的最大资源实例数。当用户申请一组资源时，系统必须确定这些资源的分配是否仍会使系统处于安全状态，如果是，就分配资源，否则，必须等待直到某个其他进程释放足够资源为止。

定义以下数据结构

- Available：表示每种资源的现有实例数量
- Max： 表示每个进程的最大需求
- Allocation： 每个进程限制所分配的各种资源类型的实例数量
- Need：每个进程还需要的剩余资源

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

安全性算法

确定计算机系统是否处于安全状态

- 定义变量 `Work = Available， finish[i] = false;`
- 查找满足 `finish[i] = false && Needi <= Work` 的 `i` ， 如果没有，就跳转到第 4 步
- `Work += Allocation; finish[i] = true` , 返回第 2 步
- 如果所有的 `i` 满足 `finish[i] = true` 则系统安全

一共需要 $m \times n^2$ 的操作

资源请求算法

- 如果 $Requset_t \leq Need_i$ 跳到第二步，否则出错
- 如果 $Requset_t \leq Available$ 跳到第三部，否则需要等待
- 如果可以分配，那么修改状态
    - $Available = Available - Request_i$
    - $Allocation_i = Allocation_i + Requset_i$
    - $Need_i = Need_i - Request_i$

死锁检测

- 允许系统进入死锁状态
- 检查系统状态从而确定是否出现锁的算法
- 从死锁状态恢复的算法

每种资源类型只有单个实例

等待图: 从资源分配图中，删除所有资源类型节点， 合并适当边。

当且仅当等待图中有一个环，系统存在死锁。

每种资源类型可有多个实例

类似于银行家算法，只有有一个不满足就是处于死锁状态

应用检测算法

何时调用检测算法，取决于

- 死锁可能发生频率
- 死锁发生时受影响进程的数量

如果进程发生死锁，就应该经常调用检测算法。

死锁恢复

- 终止一个或多个进程
- 从一个或多个死锁进程那里抢占一个或多个资源

进程终止

- 终止所有死锁进程
- 一次只终止一个，直到取消死锁循环为止（开销较大），应该终止代价最小的进程，判断因素：
    - 进程优先级
    - 进程计算了多久，还需多久
    - 使用了多少，什么类型的资源
    - 需要多少资源能完成
    - 多少进程需要被中止
    - 进程是交互的还是批处理

资源抢占

- 选择一个牺牲品（代价最小化）
- 回滚
    - 完全回滚： 终止进程并重新执行
    - 回滚到足够打破死锁
- 饥饿
    - 确保一个进程只能有限地被选择为牺牲品

总结

三种处理死锁的方法:

- 使用一些协议来预防或避免死锁，确保系统不会进入死锁状态
- 允许系统进入死锁状态，检测死锁，并恢复
- 忽略这个问题

当且仅当 4 个必要条件同时成立才会发生死锁， 确保则 4 个必要条件不成立可以预防死锁

死锁避免算法比预防算法要求低

检测到死锁，系统通过终止某些死锁进程或抢占某些死锁进程的资源来从死锁中恢复

抢占时需要考虑： 选择一个牺牲品、回滚和饥饿

第三部分 内存管理

第 8 章 内存管理

背景

- 基本硬件概述
- 符号内存地址到实际物理地址的绑定
- 逻辑地址与物理地址的差别
- 动态装载
- 动态链接代码及共享库

基本硬件

CPU 在使用数据前必须先把数据移到内存中

然而 CPU 访问内存通常需要多个时钟周期，内存访问频繁，经常需要暂停 (stall)，因此加入了高速缓存 (cache)

内存需要确保操作系统不被用户进程访问，可通过硬件实现。

确保每个进程都有独立的内存空间，需要确定进程可访问的合法地址的范围，并确保进程只访问其合法地址。通过基地址寄存器和界限地址寄存器实现。

- 基地址寄存器 base register：最小的合法物理内存地址
- 界限地址寄存器 limit register： 范围的大小

内存空间保护：通过 CPU 硬件对用户模式所产生的每一个地址与寄存器的地址进行比较。当用户模式下执行的程序试图访问非自己的内存，就会当成错误处理。

只有操作系统在内核模式下才能修改这两个寄存器

地址绑定

输入队列 (input queue)： 在磁盘上等待调入内存以执行的进程。

- 编译时 (compile time): 在编译时就知道进程在内存中驻留地址，生成绝对代码。如 DOS 的 COM 程序
- 加载时 (load time)：可重定位代码 (relocatable code)， 绑定会延迟到加载时 [静态地址重定位]
    - 无须硬件支持
    - 重定位以后不能在内存中移动
    - 要求存储空间是连续的
- 执行时 (execution time)：进程可以在执行时从一个内存段移到另一个内存段，绑定延迟到执行时执行，绝大多数操作系统采用的方法（需要硬件支持，mmu）[动态地址重定位]

逻辑地址空间与物理地址空间

逻辑地址 (logical address)： CPU 所生成的地址（也称虚拟地址）

物理地址 (physical address)：内存单元所看到的地址（加载到内存地址寄存器(memory-address register) 中的地址）

编译和加载时绑定方法中：逻辑地址 = 物理地址

执行时地址绑定：逻辑地址!= 物理地址

逻辑地址空间 (logical address space)：由程序生成的所有逻辑地址的集合

物理地址空间 (physical address space)：逻辑地址相对应的所有物理地址的集合

内存管理单元 (memory-management unit, MMU)： 管理运行时从虚拟地址到物理地址的映射。

基地址寄存器作为重定位寄存器 (relocation register)， 用户进程所生成的地址在交送内存之前，都将加上重定位寄存器的值。

用户程序处理逻辑地址，内存映射硬件将逻辑地址转变为物理地址。

动态加载

一个子程序只有在调用时才被加载。

优点：

- 不用的子程序不会被加载。
- 更好的内存占用
- 不需要操作系统的特殊支持

覆盖技术：

先后调入程序段到同一内存的空间

动态链接与共享库

静态链接，由加载程序合并到二进制程序镜像中

动态链接：

- 装入时链接

        - 使用动态库导入表
- 运行时链接
    - 使用 API 装入动态库
    - 优点：
        - 共享
        - 部分装入
        - 便于局部代码修改
        - 便于运行环境适应
    - 缺点：
        - 链接开销
        - 管理开销

被链接的共享代码称为动态链接库 (DLL, Dynamic-Link Library) 或共享库(shared library)

二进制镜像对每个库程序的应用都有个存根 (stub)， 用于指出如果定位适当的内存驻留库程序，如果程序不在内存如何装入内存。

优点： 占用体积小， 方便更新。

交换

进程暂时从内存中交换 (swap) 到备份存储 (backing store) 上，当需要再次执行时调回到内存中。

备份存储 (backing store)：快速大容量的存储设备，可以直接访问

滚出和滚入：内存管理器交换出低优先级的进程， 装入高优先级进程

交换时间的主要部分是传输时间

总传输时间与内存交换量成正比。

连续内存分配

contiguous memory allocation

内存通常分为两个部分：

- 预留给操作系统的低地址部分（含有中断向量表）
- 用户进程的高地址部分

内存映射与保护

通常使用重定位寄存器和界限地址寄存器实现

重定位寄存器含有最小的物理地址值

界限地址寄存器含有逻辑地址的范围值

MMU 动态地将逻辑地址加上重定位寄存器的值映射为物理地址，再移交给内存单元

重定位机制为操作系统提供了灵活性，可以动态改变操作系统大小（驱动的加载）

内存分配

多分区方法：将内存分为多个固定大小的分区 (partition)，每个分区只能容纳一个进程

当一个分区空闲，就从输入队列中选择一个进程，调入到空闲分区

可变分区：使用一个表记录哪些内存可用和已占用。一大块可用内存称为孔 **(hole)**

动态存储分配问题：

常用方法： 首次适应 (frist-fit)、最佳适应 (best-fit)、最差适应 (worst-fit)

- 首次适应 First-fit： 分配第一个足够大孔，查找可以从头开始，也可以从上次首次适应结束时开始，一旦找到足够大的空闲孔，就可以停止
- 最佳适应 Best-fit： 分配最小足够大的孔。必须查找整个列表，可以产生最小剩余孔
- 最差适应 Worst-fit： 分配最大的孔。必须查找整个列表，可以产生最大剩余孔。

首次适应和最佳适应的执行时间和利用空间都好于最差适应。首次适应方法最快。最差适应可以预留较大的孔。

碎片

首次适应和最佳适应都会导致外部碎片问题 (当所有总的可用内存之和可以满足请求，但不连续)

50% 规则： 对采用首次适应方法，不管使用什么优化，假定由 N 个可分配块， 那么可能由 0.5N 个外部碎片，及 1/3 的内存可能不能使用

内部碎片：在分配块内的空间

解决外部碎片的一个方法： 紧缩 (compaction)

移动内存的内容，以便所有空闲空间合并成一整块。紧缩仅在重定位是动态并在运行时可采用。

最简单的合并算法: 简单地将所有进程移到内存的一段，而将所有的孔移动到内存的另一端， 以生成一个大的空闲块。这种方案开销较大

另一个解决方法: 允许物理地址空间为非连续。

分页

paging， 分页内存管理方案允许进程的物理地址空间可以时非连续的。

基本方法

将物理内存分为固定大小的块，称为帧 (frame)， 将逻辑内存页分为同样大小的块， 称为页 (page)

当需要执行进程时， 其页从备份存储中调入到可用的内存帧中。

硬件支持：

由 CPU 生成的每个地址分为两个部分，页号 (p) 和页偏移 (d)。 页号作为页表中的索引。页表包含每页所在物理内存的基地址，与页偏移组合形成物理地址。

分页没有外部碎片，但是还是有内部碎片。

分页的一个重要的特点时用户视角的内存和实际的物理内存的分离。

帧表 (frame table) 中， 保存着物理存你的分配细节， 每个条目对应一个帧。

分页增加的切换时间。

硬件支持

页表的硬件实现，最简单的方法:

将页表作为一组专用寄存器来实现。

当页表比较大的时候，将页表放在内存中，

页表基寄存器 **(page-table base register, PTBR)** 指向页表

页表长度寄存器 **(Page-table length register, PRLR)** 表明页表的大小。

不过需要两次内存访问， 内存访问速度减半。

标准解决方案： 使用小但是专业且快速的硬件缓冲，称为转换表缓冲区 **(translation look-aside buffer, TLB**, 是关联的快速内存，由键（标签）和值组成。

不过 TLB 中条目数不多，只包括页表中的一小部分，如果页码不再 TLB 中 (TLB 失效)，那么就需要访问页表，然后把页号和帧号存到 TLB 中，如果 TLB 已满， 就是用最少使用替换(LRU) 或随机替换。并且允许有些条目固定下来。

TLB 中还保存地址空间标识符 **(address-space identifier, ASID** , 可用来唯一标识进程，为其提供地址空间保护。保证当前运行进程的 ASIO 与虚拟页中 ASID 相匹配，如果不匹配就失效处理。如果不支持 ASID， 那么上下文切换的时候，TLB 就需要被冲刷 (flushed) 或删除。

页号在 TLB 中被查找到的百分比称为命中率

有效访问时间 Effective Access Time（EAT）= 访问内存时间 *命中率 + （1 - 命中率）* （访问页面 + 访问内存时间）

保护

分页环境下， 内存保护是通过每个帧相关的保护位（保存在页表中）来实现的。

使用一个位来定义一个页是可读写还是只读的

使用有效位和无效位可以捕捉到非法地址

共享页

分页的优点之一在于可以共享公共代码

页表结构

层次页表

对于非常大的页表，可以将其划分为更小部分

划分方法一： 两级分页算法，将页表再分页。逻辑地址由 {P1, P2, d} 组成， 也称为向前映射页表 **(forward-mapped page table)**

哈希页表

处理超过 32 位的地址空间常用哈希页表， 并以虚拟页码作为哈希值。

哈希表中每个元素由 3 个域:

- 虚拟页码
- 所映射的帧号
- 链表中下一个元素的指针

变种： 群集页表 (clustered page table)， 适合 64 位，对于稀疏地址空间特别有用

每一条目包括多页，一个页表条目可以存储多个物理页帧的映射。

反向页表

通常情况下， 每个进程都有一个相关页表。页表是按照虚拟地址排序的，操作系统计算出对应条目在页表中的位置，并可以直接使用该值。那样就存在一个缺点：每个页表可能有很多项，可能会消耗大量的物理内存，却仅仅用于跟踪物理内存如何使用。

反向页表, inverted page table， 对于每个真正的内存页或帧才有一个条目。每个条目包括保存在真正内存位置的页的虚拟地址以及拥有该页的进程的信息。

这种方案减少了存储每个页表所需要的内存空间，但是增加了查找页表所需要的时间。

反向页表按物理地址排序， 而查找是根据虚拟地址，因此可能需要查找整个表来寻求匹配。可以使用哈希页表 + TLB 来优化。

而且反向页表实现共享内存比较困难，不能实现被映射到一个物理地址多个虚拟地址。

分段

分页管理内存的问题：用户视角的内存和实际物理内存的分离。

基本方法

分段 (segmentation) 是支持用户视角的内存管理方案。逻辑地址空间是由一组段组成的。每个段都有名称和长度。地址指定了段名称和段内偏移。

用户通过段名称和偏移来指定地址。

硬件

段表 (segment table) 实现将二维的用户定义定制映射为一维的物理地址。段表的每个条目都有段基地址（**STBR**）和段界限**(STLR)**。段基地址包含该段在内存中开始的物理地址，段界限指定该段的长度。

逻辑地址由：段号 s 和段内偏移 d 组成。

MULTICS 系统通过分页来解决外部碎片和冗长搜索时间的问题。

段表条目不包含段的基地址，而是包含此段的页表的基地址。

第 9 章 虚拟内存

背景

能够执行只有部分在内存中的程序的好处：

- 程序不再受现有的物理内存空间的限制，用户可以为一个巨大的虚拟地址空间编写程序，简化了变成工作量
- 每个用户程序使用了更少物理内存，更多的程序可以同时运行，增加了 CPU 使用率。
- 载入或交换每个用户程序到内存内所需的 I/O 变少。

虚拟内存 (virtual memory) 将用户逻辑内存与物理内存分开。

进程的虚拟地址空间就是进程如何在内存中存放的逻辑 (或虚拟) 视图

随着动态内存的分配，对可以向上生长，随着子程序的不断调用，栈可以向下生长，堆栈之间的巨大空白空间为虚拟地址的一部分。称为稀地址空间。

虚拟内存也允许文件和内存通过共享页而为两个或多个进程所共享。优点：

- 将共享对象映射到虚拟地址空间，系统库可为多个进程所共享
- 虚拟内存允许进程共享内存
- 允许在用系统调用 `fork` 创建进程期间共享页，从而加快进程的创建

按需调页

demand paging, 在需要的时时候才调入相应的页。

好处:

- 更少的 I/O
- 更少的内存需求
- 更快的响应
- 更多的用户

按需调用类似于使用交换的分页系统，进程驻留在第二季存储器上。使用懒惰交换 (lazy swapper)

区别：

- 交换对整个进程进程操作
- 调页程序 (paper)，只对进程的单个页进程操作。

基本概念

换入进程时，调页程序推测在该进程再次换出之前会用到哪些页。把必须的页调入进程，避免了读入不使用的页，减少了交换时间和所需物理内存空间。

使用有效 **-** 无效位 **(valid0invalid bit)** 区分哪些页在内存里。

当进程视图访问尚未调入到内存的页， 就会产生页错误陷阱 **(page-fault trap)**。

处理错误的程序：

- 检查进程的内部页表，以确定该引用是合法还是非法的地址访问
- 如果引用非法，那么就中止进程。如果引用有效但是尚未调入页面，那么现在调入
- 找到一个空闲帧
    - 如果没有空闲帧，使用页面置换
- 调度一个磁盘操作，一般从所需要的页调入刚分配的帧
- 当磁盘读操作完成后， 修改进程的内部表和页表，以表示该页已经在内存中。
- 重新开始因陷阱而中断的指令。

纯粹按需调页 **(pure demand paging)**

一开始所有的页都不在内存中，只有在需要时才将页调入内存。

程序应具有局部引用 **(locality of reference)**，才能使得按需调页性能较为合理。

支持按需调页的硬件:

- 页表：通过有效 - 无效位或保护位的特定值，将条目设为无效
- 次级存储器：保护不再内存中的页。称为交换空间 **(swap space)**

请求调页的关键要求：在页错误后重新执行指令。

主要困难：一个指令可能改变多个不同的位置或者源和目的块由重叠。当指令执行到一半的时候，页错误后不能简单地再次执行该指令。

解决方案:

- 微码计算并试图访问两块的两端，先检测是否有页错误再执行移动。
- 使用临时寄存器来保存覆盖位置的值

按需调页的性能

有效访问时间 = $(1-p) \times ma(\text{内存访问时间}) + p \times \text{页错误时间}$

主要页错误处理时间:

- 处理页错误中断
- 读入页
- 重新启动进程

按需调页的另一个重要方面是交换空间的处理和使用。如果进程开始将整个文件镜像复制到交换空间，并从交换重建执行按页调度，效果会更好。也可以从文件系统中按需调页到交换空间。

写时复制

Copy-on-Write, COW

这种方法允许父进程与子进程开始时共享同一页面。这些页面标记为写时复制页， 即如果任何一个进程需要对页进行写操作，那么就创建一个共享页的副本

从空闲缓冲池分配空闲页，用按需填零 **(zero-fill-on-demand)** (在分配前先填 0，清除以前的内容) 分配页。

`vfork()`：将父进程挂起， 子进程使用父进程的地址空间。

页面置换

如果增加了多道程序，那么会过度分配 (over-allocating)

操作系统可以交换出一个进程，释放其所有帧，降低多道程序的基本。

基本页置换

- 查找所需页在页面上的位置
- 查找一个空闲帧
    - 如果有空闲帧，使用它
    - 如果没有，那么就使用页置换算法选择一个牺牲帧 **(victim frame)**

        - 将牺牲帧写入磁盘，改变页表和帧表
    - 将所需页读入新的空闲帧，改变页表和帧表
    - 重启用户进程

可以通过修改位 **(modify bit)** 或脏位 **(dirty bit)** 以降低额外开销。以决定牺牲帧是否需要写到磁盘上去。

需要帧分配算法 **(frame-allocation algorithm)** 和页置换算法 **(page-replacement algorithm)**

置换算法：通常使用最小页错误率的算法

FIFO 页置换

最简单的页置换方法。

Belady 异常 (Belady’s anomaly)：对有的页置换算法，页错误率可能会随着所分配的帧数的增加而增加。

FIFO 算法容易实现，但是所替代的页可能包含一个以前初始化并且不断使用常用变量。

最优置换

optimal page-replacement algorithm

搜索所产生的页错误率是最低的，且绝没有 Belady 异常问题。

称为 OPT 或 MIN，置换最长时间不会使用的页。

难点：需要引用串的未来知识。

LRU 页置换

最近最少使用算法 (least-recently-used(LRU) algorithm)

LRU 选择最长时间没有使用的页来置换。

实现方法：

- 计数器：为每个页表项关联一个使用时间域，并为 CPU 增加一个逻辑时钟或计数器。每次内存引用，计数器都增加，并且时钟寄存器内容会被复制到相应页所对应页表项的使用时间域内。置换具有最小时间的页。每次都需要搜索页表并且写入内存，页表改变页需要保持时间。
- 栈：每引用一个页，就该页从栈中删除并放到顶部。

没有 Belady 异常，和最优置换属于同一类算法 (栈算法)

如果每次引用都需要中断，就使得内存引用慢至少 10 倍，因此需要硬件支持。

近似 LRU 页置换

LRU Approximation Algorithms

使用引用位

附加引用位算法

为位于内存内的每个表中的页保留一个 8 位的字节，再规定时间间隔内，将引用位转移到 8 位字节的高位，然后其他位右移。8 位字节的无符号整数最小值的页为 LRU 页，可以被置换。

如果历史位降为 0，就是第二次机会页置换算法 **(second-chance page-replacement algorithm)**

二次机会算法

基本算法还是 FIFO 算法，当引用位为 1 时，给该页第二次机会，并将引用位清零。

如果所有位被设置为 1，那就是 FIFO 置换

一种实现方法就是使用循环队列。

增强型二次机会算法

使用引用位和修改位作为有序对

- (0, 0): 用于置换最佳页
- (0, 1): 最近没有使用但是修改过，不是很好
- (1, 0): 最近使用过但是没有修改，可能很快被使用
- (1, 1)

检查页的类型，置换最低非空类中的页。

给修改过的页更高的级别，降低所需 I/O

基于计数的页置换

最不经常使用页置换算法

least frequently used(LFU) page-replacement algorithm

置换计数最小的页，并且定期将次数寄存器有移，指数衰减平均使用次数

最常使用页支援算法

most frequently used(MFU)page-replacement-algorithm

具有最小次数可能刚刚调进来，还没有使用。

页缓冲算法

系统通常保留一个空闲帧缓冲池，无需等待牺牲帧写出，就可以尽可能快地重启。

或者保留一个空闲帧池，发生页错误时可以先搜索空闲帧池，此时不需要 I/O

应用程序与页置换

有的操作系统允许特殊程序将磁盘作为逻辑块数组使用，不需要通过文件系统，称为生磁盘 **(raw disk)**，对数组的 I/O 称为生 **I/O**

应用程序可以使用适合自己的置换模式。

帧分配

帧的最少数量

帧分配策略受到的限制:

- 所分配的帧不能超过可用帧的数量
- 必须分配至少最少数量的帧
    - 性能
    - 必须有足够的帧来容纳所有单个指令所引用的页

每个进程帧的最少数量是由体系结构决定，而最大数量是由物理内存的数量来决定的。

分配算法

- 固定分配 fixed allocation
    - 平均分配 (equal allocation)： n 个进程之间分配 m 帧， 每个进程 $\frac{m}{n}$ 帧
    - 比例分配 (Proportional allocation)：根据进程大小，将可用内存分配给每个进程。
- 优先级分配 priority allocation
    - 一开始使用比例分配
    - 如果进程发生页错误，可以从低优先级的进程选择帧置换

全局分配与局部分配

全局置换：从所有帧集合中选择一个置换帧（问题：进程不能控制其页错误率，但是具有更好的系统吞吐量，可以使用其他进程不常用的内存）

局部置换：从自己分配帧中进行选择

系统颠簸

Thrashing， 进程没有足够的页，发生频繁的页调度行为，在换页上用的时间要多于执行时间，这个进程就在颠簸

系统颠簸的原因

颠簸导致严重的性能问题

当 CPU 使用率比较低的时候，CPU 调度程序就会增加多道程序的程度，就会引起更多的页错误。当继续增加多道程序的程度，就会出现系统颠簸，系统吞吐量陡降，页错误增加，有效内存访问时间增加。

局部置换算法（或优先置换算法）可以限制系统颠簸。但是即使没有颠簸，也会增加其有效访问时间。

工作集合策略研究一个进程实际使用多少帧，定义了进程执行的局部模型（**locality model**）

局部模型说明当进程执行时，从一个局部移到另一个局部。局部是一个进程使用页的集合。一个程序通常由多个不同局部组成。

如果分配的帧数小于现有的局部大小，系统就会颠簸。

工作集合模型

working-set model，是基于局部性假设的。

最近引用的页集合称为工作集合 **(working set)**

工作集合窗口 (working-set window)

- 如果太小则不能包含整个局部
- 如果太大可能包含多个局部
- 如果为无穷大，那么就为进程执行所接触的所有页的集合

总的帧需求量 = 每个进程工作集合的和

如果总需求大于可用帧的数量，那么有的进程就得不到足够的帧，系统就会出现颠簸。

工作集合策略：系统跟踪每个进程的工作集合，并为进程分配大于其工作集合的帧数。

- 有空闲帧：启动另一进程
- 所有工作集合之和大于总的可用帧数：暂停一个进程

防止了颠簸，并尽可能提高多道程序程序，优化了 CPU 使用率

困难：跟踪工作集合。

使用中断和引用位（历史位），判断是否在工作集合。

页错误频率

page-fault frequency, PFF

防止颠簸更为直接的方法。

通过页错误频率判断是否需要帧。

如果实际错误率低，那么就放弃一些帧

如果实际错误率高，那么就获得一些帧

内存映射文件

将文件 I/O 作为普通内存访问，称为文件的内存映射 (memory mapping)

开始的文件访问按普通的请求页面调度来进行，会产生页错误。

简化了文件访问和使用

允许多个进程将同一文件映射到各自的虚拟内存中，以允许数据共享

内核内存的分配

内核内存分配通常是从空闲内存池中获取的，而不是从满足普通用户模式进程的内存链表中获取。

- 内核需要为不同大小的数据结构分配内存
- 用户进程所分配的页不必要在连续的内存，而有的硬件需要

其他考虑

预调页

prepaging

试图阻止大量的初始调页。

预调页成本应该小于处理相应页错误的成本

页大小

总的来说趋向大的页

内部碎片和局部性需要小页

表大小和 I/O 时间需要大页

TLB 范围

TLB 命中率 hit ratio：通过 TLB 而不是页表进行的虚拟地址转换的百分比

增加 TLB 条数可以增加命中率

TLB 范围：通过 TLB 可以访问的内存量，等于 TLB 条数和页大小的积

可以通过增加页的大小或者提供多种页大小增加 TLB 范围

反向页表

可以降低为了跟踪虚拟地址到物理地址转换所需的物理内存的数量

程序结构

数据结构和程序结构的仔细选择可以增加局部性，避免过多的页错误

编译器和载入器对调页也有重要影响

I/O 互锁

I/O Interlock

有时需要允许有些页在内存中被锁住（当需要对用户的内存进行 I/O 时）

解决方法：

- 不对用户内存进行 I/O，在系统内存和 I/O 设备之间进行，会带来高开销
- 允许页锁在内存中，每个帧有一个锁住位

第四部分 存储管理

第 10 章 文件系统接口

文件系统的组成：

- 文件（存储相关数据）
- 目录结构（组织系统内的文件并体哦概念股有关文件的信息）

文件概念

记录在外存上的相关信息具有名称的集合

连续的逻辑地址空间

类型：

- 数据
- 程序

文件结构：

- 二进制
- 简单的记录结构
    - 线性
    - 定长
    - 可变长度
- 复杂的结构
    - 格式化的文档
    - 动态加载的文件

文件属性

通常包括属性：

- 名称 Name
- 标识符 Identifier
- 类型 Type
- 位置 Location
- 大小 Size
- 保护 Protection
- 时间、日期和用户标识 Time, data and user identification

文件信息保存在目录结构中。

文件操作

- 创建文件 Create：
    - 为文件在文件系统中找到空间
    - 在目录中创建一个条目
- 写文件 Write
- 读文件 Read
- 在文件内重定位 Reposition within file
- 删除文件 Delete
- 截短文件 Truncate：
    - 删除内容但保留属性

系统维护一个包含所有打开文件的信息表 (打开文件表， open-file table)

打开文件的信息：

- 文件指针
- 文件打开计数器
- 文件磁盘位置
- 访问权限

文件还有加锁机制

- 强制 Mandatory：系统阻止其他进程访问已加锁的文件

- 建议 Advisory：在访问文件之前需要取得锁

文件类型

UNIX 系统采用幻数 (magic number)(保存在文件的开始部分) 大致表明文件类型。也可以根据扩展名。

文件结构

用于表示文件的内部结构

内部文件结构

一般会将若干个逻辑记录打包，再放入物理记录。

访问方法

顺序访问

文件信息按顺序，一个记录接着一个记录低加以处理

适用于顺序访问设备和随机访问设备

直接访问

文件由固定长度的逻辑记录组成，以允许程序按任意顺序进行快速读写。常用于访问大量信息（数据库）

其他访问方式

通常涉及创建文件索引，索引包括各块的指针。

目录结构

存储结构

磁盘可以整体地用于一个文件系统，也可以每一个分区创建一个文件系统，也可以多个磁盘创建一个分区。

一个分区分为目录和文件

目录概述

目录可以看作符号表，他能将文件名称转换成目录条目。

目录的操作：

- 搜索文件
- 创建文件
- 删除文件
- 遍历目录
- 重命名文件
- 跟踪文件系统

目标:

- 效率： 快速定位文件

- 命名：
    - 两个用户对于不同的文件可以有相同的名字
    - 相同的文件可以有不同的名字
- 分组
- 多用户系统的共享

单层结构目录

最简单的目录结构，所有文件都包含在同一目录，便于理解和支持。

双重结构目录

主文件目录 (master file directory, MFD)

为每个用户创建独立目录（user file directory, UFD）

对用户进行了隔离，解决了名称冲突的问题

缺点：用户需要某个任务上进行合作和访问其他文件的要求，还有系统文件的共享

解决方法：修改搜索步骤，设置一个搜索路径（包括系统文件）

树状结构目录

将目录结构扩展为任意高度的树

目录包括一组文件和子目录

路径名：绝对路径和相对路径

- 绝对路径名：从根开始并给出路径上的目录名直到所指定的文件
- 当前目录名：从当前目录开始定义路径

用户初始当前目录在用户进程开始或者用户登陆时指定

允许用户自定义自己的子目录结构，按一定结构组织文件

还能访问其他用户的路径

无环图目录

树状结构禁止共享文件和目录，无环图 (acyclic graph) 允许共享子目录和文件。

无环图是树状结构的扩展

实现共享文件和目录的方法：

- 创建一个称为链接的新目录条目（特殊的类型或目录条目格式），实际上是另一文件或目录的指针。
- 重复所有共享文件的信息，修改文件时需要维护一致性。

存在的问题：

- 一个文件可以有多个绝对路径名。对于遍历操作需要注意重复。
- 删除会留下悬挂指针指向不再存在的文件

对于链接，一般的做法为，如果原文件被删除，其符号链接并不删除。

删除的另一方法：保留文件直到删除其所有引用为止。需要为每个文件保留一个引用列表或计数。

通用图目录

当树状结构目录增加链接时，树状结构就会被破坏，产生简单的图结构。（可能存在环）

问题：

- 遍历可能会死循环或者多重遍历
- 无法确定什么时候可以删除

避免多次搜索同一部分的解决方法：

- 限制搜索时访问目录次数
- 遍历目录时避开链接

当存在自我引用时候，引用计数不可能为 0，因此需要垃圾收集，但是极为费时。

垃圾收集方案：遍历整个文件系统，并标记所有可访问的空间，第二次遍历将没有标记的收集到空闲空间链表上。（标记方法可以确保只进行一次遍历）

如何使得无环：

- 只允许链接到文件
- 垃圾回收
- 新建链接得时候检测环

文件系统挂载

文件系统在被系统上的进程使用之前必须挂载 (mount)

挂载步骤：

- 确定挂载点 (mount point)， 通常为空目录
- 验证设备上是否存在有效文件系统（通过驱动读取目录验证）

不同系统的结果：

- 不允许在包含文件目录下挂载
    - 或使已存在的文件不可见，直到文件系统被卸载
- 允许多次重复挂载
    - 或只允许挂载一次

文件共享

多用户

绝大多数多用户系统采用文件（或目录）拥有者（或用户）和组的概念。

拥有着：目录最高控制权的用户，可以改变属性和授权访问。

组：对文件拥有相同权限的用户子集。

拥有者 ID 和组 ID 于文件属性一起保存

远程文件系统

远程文件共享方式：

- 通过程序（如 ftp）实现文件的人工传输
    - 匿名或验证访问
- 分布式文件系统 (DFS)，远程目录可本机直接访问
    - 紧密结合
- 万维网，使用浏览器访问
    - 几乎总是匿名

客户机 - 服务器模型

- 服务器可以服务多个客户端
- 验证一般不太安全，可能会被欺骗
- UNIX 的网络文件系统 NFS
- Windows 的标准协议 CIFS

分布式信息系统

也称分布式命名服务 distributed naming services

提供用于远程计算所需信息的同一访问，域名系统 (DNS) 为整个 Internet 提供了主机名称到网络地址的转换

故障模式

恢复故障需要在客户机和服务器之间一定的状态信息

无状态的 NFS 在每个请求包含了所有信息，恢复很简单但是不安全

一致性语义

consistency semantics

是评估文件系统对文件共享支持的一个重要准则

规定了一个用户所修改的数据何时对另一用户可见。

在 `open()` 和 `close()` 操作之间的一系列访问称为文件会话

一致性语义：

- UNIX 语义
    - 一个文件于单个物理映射相关联，是互斥资源
    - 一个用户对文件的修改可以理解被其他用户看到
    - 允许多用户共享文件指针
- 会话语义（AFS）
    - 一个文件同时可与多个物理映射暂时相关联，允许并发读写
    - 一个用户对文件的写不能立即被其他用户看到
    - 一旦文件关系，对其的修改只能被以后打开的会话看到
- 不可修改共享文件语义
    - 只能读

保护

使信息不受物理损坏（可靠性）和非法访问（保护）

可靠性由文件备份提供。

保护有多种方法。

访问类型

如果系统允许对其他用户文件进行访问，那么就需要文件保护

控制访问 controlled access

控制的操作类型:

- 读
- 写
- 执行
- 添加
- 删除
- 列表清单

访问控制

解决保护问题的常用方法：根据用户身份进行控制

实现基于身份访问的普通方法：增加一个访问控制列表（access-control list，ACL），给定每个用户名及其允许的访问类型

缺点:

- 当不知道用户列表时，创建列表会比较麻烦
- 目录条目必须可变大小，不利于空间管理

改进:

为拥有着、组、其他创建访问列表

三种访问模式:

- Read
- Write
- Execute

三种用户类型:

- Owner access
- Group access
- Public access

其他保护方式

为每个文件加上密码

第 11 章 文件系统的实现

文件系统结构

磁盘的特点：

- 可以原地重写
- 可以直接访问磁盘上任意一块信息

文件系统的设计问题：

- 如何定义文件系统对用户的接口
- 创建数据结构和算法将逻辑文件系统映射到物理外存设备

分层设计的文件系统：

- 应用程序
- 逻辑文件系统
  - 管理元数据（文件系统的所有结构数据，不包括实际数据）
  - 根据符号文件名管理目录结构，提供文件组织模块信息
  - 通过文件控制块（file control block，FCB，包括文件的信息）维护文件结构
  - 负责保护和安全
- 文件组织系统
  - 将逻辑块（簇）地址转换成物理块地址
  - 空闲空间管理器，跟踪未分配的块并根据要求提供文件组织模块
- 基本文件系统
  - 向驱动发送一般命令，对磁盘上物理块进行读写
- I/O 控制
  - 最底层
  - 由设备驱动程序（翻译器、控制硬件控制器）和中断处理程序组成
  - 实现内存与磁盘之间信息传输
- 设备

文件系统实现

概述

引导控制块 (boot control block)：

- 包括系统从该卷引导操作系统所需要的信息
- UFS 中的引导块 **(boot block)**
- NTFS 中的分区引导扇区 **(partition boot sector)**

卷控制块 (volume control block)

- 包括卷（或分区）的详细信息

- UFS 中的超级块 **(superblock)**
- NTFS 中的主控文件表 **(Master File Table)**

目录结构

- 组织文件
- UFS 包含文件名、相关索引节点 (index) 号
- NTFS 存储在 MFT 中

FCB

- 文件的详细信息
- UFS 中的索引节点 (inode)
- NTFS 在 MFT 中

内存内信息

- 管理文件系统、通过缓冲提高性能
- 包括
    - 安装表：所有安装卷的信息
    - 目录结构缓存，保存近来访问的目录信息（或指针）
    - 系统范围内打开文件表 (system-wide open-file table)：每个打开文件 FCB 副本和其他信息
    - 单个进程的打开文件表 (pre-process open-file table)：指向系统范围打开文件表的指针

访问打开文件表的索引：

- UNIX 的文件描述符 **(file descriptor)**
- Windows 的文件句柄 **(file handle)**

大多数系统在内存保留了打开文件的所有信息（除了实际数据块）

分区与安装

分区：

- raw，生的，原始的，没有文件系统
- cooked，熟的，含有文件系统

生分区（raw disk）：

- UNIX 交换空间
- 数据库
- RAID 磁盘系统

根分区 (root partition): 包括操作系统内核或其他系统文件，引导时装入内存

Windows 将卷装入到独立名称空间中，用字母和冒号表示

UNIX 将文件系统装在任何目录上

虚拟文件系统

Virtual File System， VFS

文件系统实现包括三个主要层次：

- 第一层为文件系统接口
- 第二层为虚拟文件系统 (VFS) 层
    - 定义一个清晰的接口，将文件系统的通用操作和具体实现分开
    - 提供在网络上唯一标识一个文件的机制
    - VFS 可以区分本地 / 远程、不同文件系统类型的不同本地文件
- 第三层是不同的文件系统

Linux 中 VFS 的主要对象类型:

- 索引节点对象 (inode object)，表示一个单独的文件
- 文件对象 (file object)，表示一个打开的文件
- 超级块对象 (superblock object)，表示整个文件系统
- 目录条目对象 (dentary object)，表示一个单独的目录条目

目录实现

线性列表

使用存储文件名和数据块指针的线性列表

编程简单

缺点：

- 运行费时
- 查找文件需要线性搜索

哈希表

根据文件名得到一个值，并返回一个指向线性列表中元素的指针

可以使用 chained-overflow 哈希表

分配方式

连续分配

Contiguous Allocation

每个文件在磁盘上占有一组连续的块

优点：

- 简单，只需要开始位置和长度
- 支持随机访问

缺点：

- 浪费空间

- 文件不能增长（使用扩展，但是存在外部和内部碎片问题）
- 难以为新文件找到空间（使用动态存储分配）
- 存在外部碎片（可以通过合并解决，需要停机操作）

链接分配

解决了连续分配的所有问题。磁盘块分布在磁盘的任何地方。

目录包括文件第一块指针和最后一块的指针

缺点：

- 只能有效地用于文件的顺序访问，不能有效支持文件的直接访问
- 指针需要空间
- 可靠性（可能导致错误指针）
- 文件可能分散到很多柱面，寻道时间、次数增加

解决方法：

- 多个块组成簇，按簇分配而不是按块分配（增加了内部碎片）

变种：FAT，文件分配表

每个卷开始用于存储该 FAT，采用缓存，改善了随机访问时间。

索引分配

通过把所有指针放在一起，使用索引块（磁盘块地址的数组）

支持直接访问，没有外部碎片。

缺点：浪费空间（索引块需要占用一块）

索引块大小问题：

- 链接方案 ： 将多个索引块链接起来
- 多层索引：用第一层索引块指向一组第二层索引块。
- 组合方案：UFS 中，前 12 个指向直接块，最后三个指向间接块
    - 第一个为一级间接块，为索引块，包含数据块的地址
    - 第二个为二级间接块，包括一级索引块的地址
    - 第三个为三级间接块
    - 如果块大小为 4K，那么不超过 48K 的文件就可以直接访问

Linux 下目录结构

基本文件目录 (BFD)，i 节点表，存储地址

主目录 (MFD) 存储文件名和对应 BFD 中的 ID

SFD 也是一样

优点：

- 便于共享

- 检索速度快，减少了访问磁盘的次数

Linux 文件卷存储结构

- 引导块 - 超级块（文件资源表， 专用块） - 索引节点块（基本文件目录, **BFD**） - 数据块

空闲空间管理

为了记录空闲磁盘空间，系统需要维护一个空闲空间链表 **(free-space list)**

它记录了所有空闲磁盘空间（未分配给文件或目录的空间）

位向量

空闲空间表通常实现为位图或位向量

优点：

- 查找第一个空闲块和连续空闲块简单高效

需要位向量存在内存中才有高效率，因此需要额外的空间

多次分配和回收可以集中起来写盘

链表

将所有空闲磁盘块用链表连接起来，将第一空闲块支持保存在磁盘特殊位置和内存中。

缺点：遍历整个表时，效率不高，需要大量 I/O

每一次分配和回收都需要 I/O

组

将 n 个空闲块地址存在第一个空闲块中，大量空闲块地址可以很快找到

成组链接法：

每一组的第一个块存储上一个组所有块的块号

一个超级块存储最后一个组的所有块的块号。

分配：用一个堆栈保存一组的所有块号，从栈顶开始分配，直到最后一个块（存储着下一组的块号），再次读磁盘，把这个块存储的块号填充到栈中。

回收：入栈、当栈满了就写到一个磁盘块中，然后再压仅栈里

可以做到一个组才读一次磁盘。

计数

空闲空间表包括每个条目的磁盘地址和数量

效率与性能

效率

磁盘空间的有效使用主要取决于所使用的磁盘分配和目录管理算法。

性能

- 缓存
    - 板载高速缓存：可以同时存储整个磁道，使用缓存存储经常使用的数据块
    - 页面缓存：使用虚拟内存技术，存储文件数据
- 优化顺序访问
    - 马上释放 free-behind：在请求下一页的时候，马上从缓存删除上一页
    - 预读 read-ahead：所请求的页和之后一些页可一起读入并缓存

恢复

一致性检查

将目录结构数据与磁盘数据块相比较，并试图纠正所发现的不一致

备份和恢复

利用系统程序将磁盘数据备份到另一存储设备

一开始使用 完全备份 (full backup)

然后使用 增量备份 (incremental backup)

基于日志结构的文件系统

所有元数据都按顺序写到日志上。

执行一个特殊任务的一组操作称为事务 (transaction)。

使用环形缓冲，写到空间末尾的时候，从头开始写

当系统崩溃时候，可以利用日志进行恢复

NFS

Network File System

用于通过局域网访问远程文件的软件系统的实现和规范

将一组互连工作站作为具有独立文件系统的机器组合

在访问远程目录之前需要先安装 (mount)

NFS 设计目标之一：允许不同机器、操作系统和网络结构的异构环境中工作。

在两种独立实现接口之间采用基于外部数据表示 (XDR) 的 RPC

安装协议

mount protocol

在客户机和服务器之间建立初始逻辑连接

服务器维护一个输出列表 (export list)，列出哪些本地文件系统允许输出安装，并允许安装他们的机器名称

NFS 协议

- 搜索目录内文件
- 读一组目录条目
- 操作链接和目录
- 访问文件属性
- 读和写文件

NFS 服务器是无状态的

路径名转换

把路径名解析为独立的目录条目或组成部分

远程操作

除了 open 和 `cloase`，在普通 UNIX 文件操作系统调用和 NFS 协议 RPC 之间，有着一对一的对应关系

NFS 实际采用了缓冲和缓存技术以提高性能

- 文件属性（索引节点信息）缓存
- 文件块缓存

第 12 章 大容量存储器的结构

大容量存储器结构简介

磁盘

磁盘为现代计算机系统提供了大容量的内存

读写头 “飞行” 于每个磁盘片的表面之上。

磁头与磁臂 (disk arm) 相连，磁臂能将所有磁头作为一个整体一起移动。

磁盘片的表面被逻辑划分为圆形磁道 (track)

磁道进一步划分为扇区 (sector)

同一磁臂位置的磁道集合形成了柱面 (cylinder)

每个磁盘驱动器有数千个同心柱面，每个磁道有数百个扇区

传输速率 (transfer rate)：驱动器和计算机之间的数据传输速率

定位时间 **(positioning time)/** 随机访问时间 **(random access time)** 由两部分组成：

- 寻道时间 (seek time)：移动磁臂到说要的柱面所需的时间（主要）
- 旋转等待时间 **(**rotational latency)：等待所要的扇区旋转到磁臂下所需时间

磁头碰撞 (head crash) 会损坏磁盘表面，不能修复。

磁盘驱动器通过一组 I/O 总线 (I/O bus) 和计算机相连，总线包括：

- EIDE: enhanced integrated drive electronics
- ATA: advanced technology attachment
- 串行 ATA: serial ATA, SATA
- USB: universal serial bus
- FC: fiber channel
- SCSI 总线

控制器 (controller) 处理总线上的输出传输：

- 主机控制器 (host controller) 是计算机上位于总线末端的控制器
- 磁盘控制器 (disk controller) 是位于磁盘驱动器内的控制器

磁带

太慢

磁带绕在轴上，向前转后向后转并经过读写头。

磁盘结构

现代磁盘驱动器可以看作一个一位的逻辑块的数组，逻辑块是最小的传输单位。

对使用常量线性速度 (constant linear velocity, CLV) 的介质，每个磁道的位密度是相同的

驱动器会增加速度保持磁头读写数据速率恒定

或者

使用恒定圆角速度 (constant angular velocity, CAV), 内磁道到外磁道的位密度要不断降低

磁盘附属

访问磁盘存储的方式：

- I/O 端口 (或主机附属存储 host-attached storage, HAS)
- 分布式文件系统的远程主机 (网络附属存储 network-attached storage,NAS)

主机附属存储

通过本地 I/O 访问存储

复杂的 I/O 结构：

- SCSI
    - 一根总线支持 16 个设备
    - 包括主机的一个控制卡 (SCSI 引导器)
    - 15 个存储设备 (SCSI 目标)
    - 每个 SCSI 目标有访问 8 个逻辑单元的能力
- FC
    - 高速串行结构

- 存储区域网络 (SAN) 的基础
- 裁定循环 (FC-AL)：可以访问 126 个设备

网络附属存储

NAS 是数据网络中远程访问的专用存储系统

缺点：存储 I/O 操作需要使用数据网络的带宽，增加了网络通讯延迟

存储区域网络

storage area network，SAN

是服务器与存储单元之间的私有网络

优势：灵活性

磁盘调度

磁盘带宽：所传递的总的字节数除以从服务请求开始到最后传递结束的时间

FCFS 调度

先来先服务算法

特定：公平

SSTF 调度

最短寻到时间优先算法 shortest-seek-time-first

选择距当前磁头位置最短寻道时间的请求来处理

基本上是一种最短作业优先 (SJF) 调度，可能导致一些请求得不到服务，而且不是最优的

SCAN 调度

磁臂从磁盘的一端向另一端移动，同时当磁头移过每个柱面时，处理位于该柱面上的服务请求。当到达另一端时，磁头改变移动方向，处理继续

有时称为电梯算法 (elevator algorithm)

C-SCAN 调度

SCAN 的变种，提供更为均匀的等待时间

当磁头移到另一端的时候，立刻返回到磁盘开始，返回时候不处理请求

LOOK 调度

类似于 SCAN，但是磁头只移动到一个方向的最远请求

C-LOOK 返回也是只是返回到最远，不是 0

磁盘调度算法的选择

SSTF: 普通而常见， 性能比 FCFS 好

SCAN 和 C-SCAN 对于磁盘负荷较大的系统会执行得更好，不可能产生饿死

性能主要依赖于请求的数量和类型。

磁盘服务请求很大程度上受文件分配方法所影响，连续分配会产生相近的请求，而链接或索引文件会产生大量的磁头移动

目录和索引页也很重要

磁盘管理

磁盘格式化

低级格式化：为磁盘每个扇区采用特别的数据结构

逻辑格式化：创建文件系统

引导块

自举程序保存正在磁盘的启动块中，位于磁盘的固定位置

坏块

磁盘控制器使用备用块逻辑低替换坏块，称为扇区备用 (sector sparing) 或转寄(forwarding)

每个柱面都留有少量的备用块，还有备用柱面，防止系统的磁盘调度算法无效

还能采用扇区滑动 (sector slipping) 来替换坏扇区，所有扇区向某个方向偏移

交换空间管理

交换空间为虚拟内存提供最佳吞吐量

交换空间的使用

- 可以保存整个进程映像
- 可以保存内存中的页

交换空间的位置

- 普通文件系统
    - 简单，但是效率低
    - 遍历目录需要过多磁盘访问（可以缓存在物理内存中）
- 一个独立的生磁盘分区上

RAID 结构

通过冗余改善可靠性

镜像 (mirroring) 是最为简单和昂贵的引入冗余的方法：复制每个磁盘

通过并行处理改善性能

数据分散：

- 位级分散：在多个磁盘上分散每个字节的各个位，
  - 每次范围可以同时读 8 倍的数据
  - 支持数量为 8 的倍数或者能除以 8 的数量的磁盘
- 块级分散：一个文件的块分散在多个磁盘

通过负荷平衡，增加了多个小访问的吞吐量

降低了大访问的响应时间

RAID 级别

RAID 档次	最少硬盘	最大容错	可用容量	读取性能	写入性能	安全性	目的	应用产业
单一硬盘	(引用)	0	1	1	1	无		
JBOD	1	0	n	1	1	无（同 RAID 0）	增加容量	个人（暂时）存储备份
0	2	0	n	n	n	一个硬盘异常，全部硬盘都会异常	追求最大容量、速度	视频剪接缓存用途
1	2	n-1	1	n	1	最高，一个正常即可	追求最大安全性	个人、企业备份
5	3	1	n-1	n-1	n-1	高	追求最大容量、最小预算	个人、企业备份
6	4	2	n-2	n-2	n-2	安全性较 RAID 5 高	同 RAID 5，但较安全	个人、企业备份
10	4	n/2	n/2	n/2	n/2	安全性高，但在同一个子组群中不能出现两颗毁损硬盘	综合 RAID 0/1 优点，理论速度较快	大型数据库、服务器

- RAID0
  - 按块级别分散的磁盘阵列
  - 没有冗余
- RAID1
  - 磁盘镜像

- RAID2
    - 内存方式的差错纠正代码结构
    - 基于奇偶位的错误检测，系统每个字节都有一个相关的奇偶位。
    - 4 个磁盘需要 3 个额外磁盘
- RAID3
    - 位交织奇偶结构
    - 磁盘控制器可以检测一个扇区是否争取读取
    - 如果一个扇区损坏，那么知道是哪个扇区，通过计算其他磁盘扇区相应位的奇偶值可以得出损坏位是 0 或 1
    - 优点
        - 只需要一个奇偶磁盘
        - 字节的读写分布在多个磁盘上，单个块的读和写是 RAID1 的 N 倍
    - 缺点
        - 计算和写奇偶开销导致写更慢
        - 可以使用非易失性随机存储器 (NVRAM)，在计算奇偶时存储块，缓存从控制器到磁盘的数据
- RAID4
    - 块交织奇偶结构
    - 磁盘可以并行读，数据和奇偶也可以并行写
    - 小于块大小的数据的写必须访问数据所在块，修改数据后写回，相应奇偶块也要更新，称为读 - 改 - 写，单个写需要 4 次磁盘访问：两次读入旧块，两次写入新块
- RAID5
    - 块交织分布奇偶结构
    - 将数据奇偶分布在所有 N+1 块磁盘上，而不是单个磁盘上
    - 每一块，一个磁盘存储奇偶，其他存储数据
- RAID6
    - P+Q 冗余方案
    - 保存额外的冗余信息防止多个磁盘出错
    - 使用差错纠正码，如 Read-Solomon 码
- RAID0+1
    - 先分散再镜像
- RAID1+0
    - 先镜像再分散

RAID 级别的选择

扩展

RAID 的问题

稳定存储实现

三级存储结构

三级存储设备

操作系统支持

性能

第 13 章 I/O 输入系统

概述

I/O 硬件

四种 I/O 方法：

- 轮询
- 中断
- DMA
- 通道

轮询

中断

中断与当前代码没有关系，异常与当前代码有关系

CPU 在执行完每条指令之后，检测 IRL（中断请求线, Interrupt-request line），如果有来自控制器的中断请求信号，CPU 就保存当前状态并且跳转到内存固定位置的中断处理程序 (interrupt-controller)

中断处理程序判断中断原因，进行必要的处理，重新恢复状态，最后执行中断返回 (return from interrupt) 指令以便使 CPU 返回中断前的执行状态

过程:

- 设备控制器通过 IRL 发送信号 raise 中断
- CPUcatch 中断并 dispatch 到 IC 中
- IC 通过处理设备请求来 clear 中断

中断特征（由 IC 硬件提供）：

- 在关键处理时， 可以延迟中断处理
- 更有效地分发中断到合适中断处理程序
- 多级中断，区分优先级

优点：可以并行

绝大多数 CPU 有两个中断请求线：

- 非屏蔽中断：处理如不可恢复内存错误等事件
- 可屏蔽中断：可以被设备控制器用于请求服务

直接内存访问

DMA direct-memory access

当 DMA 控制器抓住内存总线时， CPU 会暂时不能访问主内存。称为周期挪用 **(cycle steal)**

DVMA direct virtual-memory access 直接虚拟内存访问

可以直接实现两个内存映射设备之间的传输，无需 CPU 的干涉或使用主内存。

步骤:

- 设备驱动器被告知传递磁盘数据到地址为 X 的缓冲区
- 设备驱动器告诉磁盘控制器从磁盘传递 C 个字节到地址为 X 的缓冲区
- 磁盘控制器初始化 DMA 传输
- 磁盘控制器向 DMS 控制器发送每个字节
- DMA 控制器向缓冲器 X 传递字节，增加内存地址并减少 C 直到 C = 0
- 当 C = 0 时，DMS 中断 CPU， 通知传输完毕。

通道

一般用于服务器

可以实现多个连续段直接传输

I/O 应用接口

设备驱动程序层的作用：为 I/O 子系统隐藏设备控制器之间的差异。

- 数据传输模式
    - 字符流设备： 按一个字节一个字节地传输，如 终端
    - 块设备：以块单位进行传输， 如磁盘
- 访问方法
    - 顺序设备：按固定顺序来传输数据， 如调制解调器
    - 随机访问设备：让设备寻找到任意数据存储位置，如 CD-ROM
- 传输调度
    - 同步：按一定响应时间来进行数据传输， 磁带
    - 异步：无规则或不可预测的响应时间，键盘
- 共享
    - 共享：可以被多个进程或线程并发使用，如键盘
    - 专用：磁带
- 设备速度
    - 延迟
    - 寻道时间
    - 传输速率
    - 操作之间的延迟
- I/O 方向
    - 读写， 磁盘
    - 只读， CD-ROM
    - 只写，图像控制器

块与字符设备

网络设备

阻塞和非阻塞 I/O

阻塞 I/O：应用程序的执行被挂起，等到系统调用完成后，应用程序就移回运行队列，并在合适的时候继续执行，并能收到系统返回的值

非阻塞 I/O：不会中止程序，马上返回任何可用的数据（等于或少于所要求的，或者为 0）

异步：调用要求的传输会完整执行，返回完整的结果（将来的某个时间）。

I/O 内核子系统

I/O 调度

缓冲

用来保存两个设备之间或在设备和应用程序之间所传输数据的内存区域。

采用缓冲的理由：

- 处理数据流的生产者与消费者之间的速度差异
- 协调传输数据大小不一致的设备
- 支持应用程序 I/O 的复制语义（保证要写入磁盘的数据是系统调用发生时候的版本）

高速缓存

可以保留数据副本的高速存储器，访问要比原始数据访问高效

有时缓冲区也可以用作高速缓存

假脱机与设备预留

用来保存设备数据的缓冲区

错误处理

I/O 保护

内核数据结构

内核 I/O 子系统总结

把 I/O 操作转化为硬件

流

性能

结语

以上就是一些操作系统的四个部分的重点知识，后面有些章节还没完善，以后有空再来完善。♥

土豪通道

打赏

读书笔记 # 操作系统

UWP | Todo 应用开发 - 进阶部分

读书笔记 | 数据库管理系统 - 原理与设计