# Virtualisation

## 2

### 2.0

程序运行：

获取指令fetch

解码decode

执行execute

虚拟化 virtualization：确保系统易于使用和高效运行

将物理资源转成更为强大通用的虚拟形式

有时将操作系统称为虚拟机 virtual machine

操作系统提供了一些接口 api ，典型的操作系统有几百个系统调用 system call 有时会叫标准库 standard library

操作系统有时叫resource manager

虚拟化让许多程序运行 共享cpu

让许多程序同时访问自己的指令和数据

虚拟化cpu （virtualizing the cpu）

在只有一个处理器的情况下，将单个cpu转换成无限个cpu

操作系统承担资源管理器（resource manager）

### 2.2

每个进程会访问自己的虚拟地址空间（virtual address space）

### 2.3

并发（concurrency）

同时处理很多事情

### 2.4

持久性（persistence）

需要硬件和软件永久persistently储存数据

硬盘驱动hard drive 是长期保存信息的通用储存库

管理磁盘的软件叫文件系统file system：以高效的方式把创建的file存在系统磁盘上

### 2.5

建立一个系统，目标：

建立一个抽象 abstraction-让系统方便易于使用

提高性能 performance-最小化操作系统的开销 minimize the overhead（额外时间-指令，额外空间-内存或磁盘）

Os 和应用程序和应用程序之间有保护（protection）-隔离 isolation，进程彼此隔离

## 4

### 4.0

上下文切换 context switch

让操作系统让操作系统停止运行一个程序然后给CPU运行另一个程序

时分共享 time sharing

允许资源由一个实体使用一小段时间，然后由另一个实体使用一小段时间，资源-（CPU 或网络连接）可以被许多人共享

空分共享

资源在空间上划分给希望用它的人

· 一旦将块分配给文件，在用户删除之前，不会将它分配给其他文件

策略调度 scheduling policy

用来决定操作系统应该运行哪些程序

### 4.1

机器状态machine state

进程可以访问的内存-地址空间address space 是进程的一部分

寄存器 PC 有时叫 IP-指令指针Instruction pointer

栈指针 stack pointer 帧指针 frame pointer 管理函数参数栈，局部变量和返回地址

### 4.2

进程API

创建create

销毁destroy 强制销毁接口

等待 wait 等待进程停止

其他控制 miscellaneous control 如暂停进程然后恢复运行

状态 status 如运行多长时间

### 4.3

进程创建

将代码和所有静态数据load到内存中-需要操作系统从磁盘读取其字节，并将它们放在内存的某处

为程序的 运行时栈run-time stack 或叫 stack 分配内存(stack 存放局部变量，函数参数，返回地址)

操作系统分配内存并提供给进程

操作系统可以用参数来初始化栈-会将参数输入main()函数就是int argc ,char \*argv[]

操作系统可能为堆 heap（显示请求的动态分配数据）分配内存

操作系统会执行一些其他初始化任务，特别是io任务

跳转到main()列程

### 4.4

进程状态

运行running 进程正在处理器上运行

就绪ready 进程已经做好准备，因某种原因还不能现在运行

阻塞 blocked 例如磁盘发起io请求

### 4.5

数据结构

进程有最终状态final 允许其他进程-大多为父进程，检查返回码，并查看是否成功运行，通常成功是0（return 0;）

## 5

### 5.1

UNIX 系统创建进程-fork exec wait

如果要操作某个进程，如终止进程，要用PID-process identifier来指明

调用fork（）创建进程，会执行相同的程序

子进程会从fork（）那段开始运行

父进程中fork返回的值是子进程的PID

CPU程序调度决定哪个进程先运行

用wait让父进程去等待子进程先执行完毕

exec可以让子进程和父进程执行不同的程序

exec 会从可执行文件中加载代码和静态数据，并用它复写自己写的代码段，堆栈和其他内存空间会被重新初始化

（将当前运行的程序替换为替换为不同的运行程序，所以如果成功调用就不会有返回）

操作系统执行此程序将参数通过argv传给进程

### 5.4

这种创建做法在构建UNIX很有用

在fork之后exec之前有运行别的代码的机会

shell 是一个应用程序(终端里)，先会有提示符prompt，用户可输入指令，输入之后shell会调用fork 创建新进程，再调用exec去执行程序，调用wait等待完成。结束后会再次输出提示符，然后等待用户输入

在终端中cowsay w>text.txt

会将输出结果重定向redirect 到文件里

当完成子进程创建后，shell 调用exec之前会先关闭标准输出standard output 然后打开文件将输出打印在文件里而不是屏幕上

管道pipe()

## 6

### 6.0

构建虚拟化机制的挑战

性能 不增加系统开销

控制权 运行进程同时保留对cpu的控制-如果没有权限，一个进程可不限制地运行并接管机器或访问没有权限的消息

直接运行 direct execution

在进程列表上创建条目

为进程分配内存

将程序加载到内存中

根据argc argv初始化栈

清除寄存器

执行main（）方法直到执行return

释放进程的内存并存进程列表中删除

受限的直接运行 limited direct execution

### 6.2

用户模式 user mode

在用户漠视下，系统无法发出io请求

内核模式 kernel mode

运行代码可以做它喜欢做的事

如果希望用户执行某种特权操作

用户程序执行系统调用-允许内核小心地向用户程序暴露某些关键功能（访问文件系统，创建销毁进程，与别的进程通信）

执行系统调用

执行特殊的 陷阱trap指令-该指令同时跳入内核并将权限级别调到内核模式

执行完成后，会调用 陷阱返回 return from trap 指令- 会返回到调用程序中并将权限下降到用户模式

c中系统调用是用汇编编写的

受限直接运行协议LDE协议：

机器启动时

初始化陷阱表（内核模式）-> 硬件记住系统调用程序的地址（硬件)

开始一个进程

（内核模式）

在进程列表上创建条目

为程序分配内存

将程序加载到内存中根据argv设置程序栈

用寄存器和PC填充栈

从陷阱trap返回

｜

V

（硬件）

从内核栈恢复寄存器

转到用户模式

跳到main

｜

v

（main 程序）

运行main

调用系统调用

陷入操作系统

｜

V

（硬件）

将寄存器存到内核栈

转向内核模式

跳转到陷阱处理程序

｜

v

（内核模式）

处理陷阱

做系统调用工作

从陷阱返回

｜

v

（硬件）

从内核栈恢复寄存器

转向用户模式

跳到陷阱之后的计数器PC

｜

v

（程序）

。。。

从main返回return

通过exit（）

｜

v

（内核模式）

释放进程内存

将进程是进程列表移除

### 6.3

进程之间切换

协作cooperative方式：等待系统调用

在进程合理运行下，运行时间长的进程会主动放弃cpu控制权

yield 系统调用-将控制权交给操作系统

如果应用程序做了非法行为，也会控制转移给操作系统

非协作 操作系统进行控制

时钟中断 timer interrupt-时钟设备可以设为每毫秒中断一次，产生中断，当前运行的程序会停止

操作系统的interrupt handle 程序会运行

操作系统会获得cpu控制权-停止或启动进程

启动时，操作系统会通知硬件那些代码在发生时间中断

启动过程中，操作系统必须启动时钟-特权操作

硬件发生中断时，要为正在运行的程序保存足够的状态，为了能从trap from return 指令恢复正在运行的程序

各个寄存器会进入内核栈

保存，恢复上下文

当获得了CPU控制权，就必须决定时继续运行当前程序还是，还是切换到另一个进程-（程序调度 scheduler）

上下文切换context switch

为当前正在运行的进程保存寄存器的值（到它的内核栈）

为即将要运行的进程（从内核栈）恢复一些寄存器的值

操作系统会执行底层的汇编代码来保存当前正在运行进程的上下文

会保存register PC 当前正在运行进程的内核栈指针

会恢复register PC 然后切换内核栈

受限直接执行协议 时钟中断

启动系统

初始化陷阱列表（内核模式）-> 记住 系统调用处理程序和时间处理程序的地址（硬件）-> 启动中断时钟（内核模式）-> 启动时钟 每隔多少ms中断CPU

运行时

运行进程A（程序）

｜

v

时钟中断（硬件）

将寄存器保存到内核栈

转向内核模式

跳入陷阱处理程序

｜

v

处理陷阱（内核模式）

调用switch（）（上下文切换）

将寄存器A保存到进程结构A

将进程结构B恢复到寄存器B

从陷阱返回

｜

v

从内核栈B恢复到寄存器B（硬件）

转成用户模式

调到B的程序计数器PC

｜

v

运行进程B（程序）

问题：

在系统调用期间发生时钟中断

处理中断时发生另一个中断

解决：

在中断处理时用禁止中断 disable interrupt -处理一个中断时，不会讲其他中断交给cpu

加锁 locking-用于多处理器

## 7

### 7.0

调度策略 scheduling policy

早起用于操作管理领域

### 7.1

工作负载 workload - 简化假设，与系统的运行程序有关

比如：1，每一个工作运行相同的时间

2，。。。

### 7.2

调度指标-用来衡量某些东西的东西

比如 ： 周转时间T（性能指标 preference）

周转时间T=完成时间T-到达时间T

要让周转时间T 每次相同

完成时间T-到达时间T 每次也都要相同

性能 preference 和公平fairness 是矛盾的- 提升性能时，会阻止一些任务运行，公平就下降了

以下为调度程序

### 7.3

FIFO先进先出first in first out

问题：当有一个耗时很长工作放在最前面，那得等把第一个工作做完之后再做下一个--这个需要很长时间，称为 护航效应convoy effect

###优化周转时间

### 7.4

最短任务优先 SJF shortest job first-先运行最短任务

问题：

短时任务到达时间比长时认为慢，就会有问题-平均周转时间变长

### 7.5

最短完成时间优先 STCF shortest time-to-completion First

到短时任务到达时，程序调度可以抢占preempt长时工作，并运行另一个任务，然后稍后运行长时工作

###

### 7.6

response time

响应时间T=首次运行T-到达运行T

问题：

如何构建对响应敏感的调度程序

###优化响应时间（在没有IO情况下）

### 7.7

轮转 Round-Robin

在一个时间切片time silence里运行一个工作

然后切换队列中的下一个任务而不是运行一个任务到结束

如果是每10ms中断一次，那时间片可以是10ms的倍数

时间片太短会有问题：因为上下文切换是有开销的-不仅要保存或恢复寄存器，在CPU高速缓存 TLB 分支预测器等硬件有大量状态，一旦切换，会被重新刷新并在此引入新状态

要权衡时间片长度以便 摊销amortize 上下文切换的成本

###

### 7.8

结合IO

在IO期间不会使用CPU

在IO完成时调度程序会做出决定

会产生中断

发出IO的进程从阻塞状态转为就绪状态

可以把有需要IO的进程拆开

把需要IO的那段时间抽出来，使其变为一小块一小块的子工作

把每一个子工作视为独立工作

在进程B等待进程A的IO完成时，让进程B使用CPU-（重叠overlap）

## 8

### 8.0

MLFQ多级反馈队列 Multi-level FeedBack Queue

MLFQ要解决优化周转时间turnaround time

MLFQ希望给交互用户很好的交互体验，要降低响应时间

### 8.1

MLFQ基本规则

MLFQ有许多独立的队列Queue

一个工作只能存在一个queue中

每个queue有不同的优先级priority level

MLFQ总是优先执行较高优先级的工作

比如有一个工作要不断进行IO操作，MLFQ会保持较高的优先级

有个工作会长时间占用CPU它的优先级会降低

如果A队列的优先级>B，运行A而且不运行b

如果优先级一样，会采用RR调度

### 8.2

尝试改变优先级

规则：

工作进入系统时，放在最高优先级

#工作用完整个时间片之后会降一个优先级

#如果工作在这个世界片中释放了CPU优先级不变

缺点：

饥饿starvation问题，如果有太多工作which会释放CPU，它们会在高优先级里不断占用CPU，导致低优先级的无法获得CPU

有些程序会有意子时间片块运行玩之前，调用IO，这样优先级保持不变，这样CPU控制权大部分都是这个程序的

### 8.3

提升优先级

周期性地提升boost priority level

规则：

经过一段时间S，会将所有工作加入到最高priority level

（用来解决进程不会饿死starvation）

但如果S设置得太高，长工作会饿，S太低，交互型工作得不到CPU合适的时间比例

所以S被称为巫毒常量 voo-doo constant

### 8.4

更好的计时方式

将带有#的规则进行重写：

无论主动放弃多少次CPU，只要一旦用完在某一层的时间配额，就会降一级而不是priority level不变

（用来解决：有些程序会有意子时间片块运行玩之前，调用IO，这样优先级保持不变，这样CPU控制权大部分都是这个程序的）

### 8.5

有些调度程序会将最高priority level留给操作系统

## 9

### 9.0

比例份额proportional-share （公平份额fair-share）

调度程序的最终目标是让每个工作能获得一定比例的CPU

### 9.1

彩票调度

彩票数ticket 代表进程占用某个资源的份额

一个进程的彩票数占总彩票数的百分比就是占用资源的份额

如果processA有75彩票，process B有25彩票

那我们希望A占用CPU75%的时间，B是25%

调度程序知道彩票数（100张，No从0～99）

调度程序会从0～99抽一张彩票

中奖的彩票决定了运行A还是B

好处：

可以避免边角情况

此方法很轻量，不需要记录任何状态

快

### 9.2

彩票货币ticket currency

全局彩票为200

假设进程A和进程B各有100张票

A有两个工作，B有一个工作

进程A里有自己的货币currency一共有1000个，它的每个工作有500个货币

进程B有自己的货币currency一共有10个，它的工作有10个货币

将货币currency转为全局彩票

A 工作1:50 工作 2:50

B 工作 100

彩票转让ticket transfer

一个进程可以将自己的彩票转给另一个进程

在客户端服务端中

客户端process给服务端发送消息

为了加快服务端执行，客户端可以把彩票转给服务端

服务端执行结束后把彩票归还给客户端

彩票通胀ticket inflation

利用通胀，一个进程可以提升或降低彩票数量

用于进程之间相互信任的环境

如果进程需要更多CPU时间，就可以加彩票

这一切不需要和其他process通信

### 9.5

为工作分配彩票

### 9.6

步长调度 stride scheduling

每个工作都有自己的步长

这个值与票数值成反比

用一个比较大的数除以票数就是步长（比如ABC进程的彩票为50，100，250，那么他们的stride可能是，200，100，40）

程序每次运行后，会让这个process的计数器增加它的stride，记录总体进展

当要进行调度是会选择strip最小的

彩票调度的优势：

不需要全局状态

在stride scheduling里，如果一个新进程加进来，那它的stride是0，就会一直运行它

问题：

两种方式都不能很好的适合IO

开始的票数分配也很难

## 10

### 10.0

多处理器调度multiprocessor scheduling

多核处理器multicore 将多个CPU组装在一块芯片上

为了让程序并行执行parallel 用多线程thread-将工作分散在多个cpu上

问题：

多处理器调度multiprocessor scheduling

### 10.1

与单核CPU的区别：

对硬件缓存cache的使用

multiprocessor 之间共享数据的方式

单CPU系统中：

有个多级硬件缓存hardware cache-会让更快执行程序，与RAM互补，让CPU又大又快

如果一个程序要从内存读取一个值

第一次读数据时是在内存里，需花费许多时间

处理器会判断该数据是否会被再用

因此会把它放入CPU缓存中

如果程序在次需要相同的数据，CPU会先查cache，这样会快

cache：

基于局部性locality

时间局部性：当一个数据被访问时再不久的将来也会被访问，像循环的数据或指令本身

空间局部性：当访问地址为x的数据，可能会访问周围的数据，像数组

问题：缓存一致性cache coherence

假设在CPU1从内存读取地址A，得到D的值

程序修改D的值为Z，只是将它的缓存改为Z，内存没有

当操作系统中断此程序的运行，让CPU2重新读取A地址的值，就会有问题

硬件提供了基本解决方案-通过监控内存访问：

在基于bus的系统中，总线窥探bus snooping：

每个缓存会监听链接所有缓存和内存的总线

如果cpu发现其放在缓存的数据进行了

会作废invalidate本地副本（从缓存移除）

或update它

### 10.2

应用程序也要关心共享数据的访问

需要使用互斥原语-锁

如果用多CPU并发访问一个共享队列

如果没有锁，只有底层协议，不会得到预期的结果

要用锁来保证数据结构状态更新的原子性

如果用两个线程调用deleteNode（）函数：

线程1，2会将head的值赋在tmp中

那它们删除的值也是同一个

释放free（）也是同一个

返回的数也是同一个

可以加锁locking来解决

要互斥锁pthread\_mutex\_t m;

函数开始时用lock（&m）;

在结束时用unlock(&m);

### 10.3

缓存亲和度cache affinity

CPU1的缓存中维护许多状态，下次该进程在相同的CPU1上运行时，会变快

但在CPU2上运行时要重新加载数据

虽然有硬件可以保证缓存一致性

但尽可能让一个进程保持在同一个CPU上

### 10.4

单队列多处理器调度Single Queue multiprocessor scheduling SQMS-简单

每个CPU都可以沾一点

问题：

可扩展性scalability：如果在代码中加锁locking来保证原子性，随着单个这样的锁的增加，系统花费巨大时间在锁的开销上

亲和性：每个工作都可能会在不同的cpu上工作cache affinity降低

多队列多处理调度Multi-Queue Multiprocessor Scheduling MQMS

会有多个调度队列

每个queue可以用不同的调度规则（比如MLFQ 或 RR）

当一个工作进入系统后，系统会随机或选择较空的队列，然后将其放入此调度队列

这样，cpu就会相互独立

所有的工作都保持在固定的CPU上（cache affinity）

问题：

负载不均load imbalance

如果有三个任务进入系统，CPU1有一个任务，CPU2有两个任务

而且如果CPU1的任务做完，CPU1就没有任务，单CPU2有两个任务

解决：

用迁移migration

将一个任务从一个CPU迁移到另一个CPU

如果CPU1有一个任务，CPU2有两个，就把一个火多个任务不断迁移

如何实现解决：

工作窃取work stealing

工作量较少的队列会不定期的偷看别的队列

如果如果目标队列比源队列更满，就会窃取任务

## 13

### 13.1

早期操作系统就是一个库

有一个程序从物理地址64kb开始，并使用剩下的所有内存

### 13.2

由于机器昂贵，人们开始共享机器

多道程序multiprogramming

多个进程在给定时间内运行

当有一个进程在等待IO的时候

操作系统会切换进程

分时系统：

让一个进程占用全部内存运行一小段时间

然后停止它，将它的状态信息存到磁盘里

加载其他进程的状态信息

（太慢了）

可以把所有进程的信息状态一直放在内存里

### 13.3

需要一个easy to use 的抽象abstract-地址空间address space（运行程序看到系统中的内存）

一个进程的address space包含运行程序所有的内存状态

代码code

栈stack -来保存当前函数调用的信息，分配空间给局部变量，传递参数和函数的返回值

堆heap -管理动态分配，像C的 malloc（），C++的new 来获得内存

代码位于地址空间的顶部-因为代码是静态的

之后地址有两个区域会增长或收缩-heap在顶部（code的下面），stack在底部

heap 在用户调用malloc时向下增长

stack 在用户调用程序时向上增长

（多线程threads不会这样）

虚拟化内存virtualizing memory-在单一物理内存上为多个进程构建地址空间抽象

如果进程A尝试在虚拟地址virtualizing address 0加载操作

操作系统在硬件的支持下，确保不是加载到物理地址0，而是物理地址XXXKB（320KB）

### 13.4

virtualizing memory 系统主要目标是透明transparency

让程序感知不到内存被虚拟化的事实

另一个目标是效率efficiency：

在时间上不会是程序运行更慢

在空间上不需要额外太多内存支持虚拟化

第三目标是保护protection：

操作系统要受到保护，不受其他进程的影响

当一个进程要加载，储存，指令提取时，不能影响仍和别的进程保持操作系统

## 14

### 14.1

内存类型

栈内存：

申请和释放是编译器隐式管理的

也被称为自动automatic 内存

在C中申请栈内存：在函数中为一个变量申请空间

· 编译器会在你进入函数的时候，在栈上开辟空间

在退出函数的时候，编译器会释放内存

堆heap内存：

所有的申请和释放都由程序员显示地完成

int \*x=(int \*)malloc(sizeof(int));

编译器看到int \* 知道为整数指针分配空间

当程序调用malloc会在堆上请求整数空间

### 14.2

malloc()

需要size\_t类型参数

返回的是一个void类型指针，让程序员去处理-通过强制转换cast

### 14.3

free（）

知道何时，如何释放内存是困难的

接受一个malloc（）函数返回的指针

### 14.4

常见错误

许多语言都支持内存管理automatic memory management

当你调用类似malloc像new时

不需要调用某些东西来释放内存

因为有垃圾收集器garbage collector-找出你不再引用的值，替你释放它

忘记分配内存

在用strcpy（）时

将原字符串复制到指针目标会出现

没有分配足够内存

缓冲区溢出buffer overflow

运行会正常运行

在某些情况下，当字符串拷贝时

会在超过分配空间的尾处写入一个字节

某些情况下是无害的，但有时会有很大的危害

malloc总是会分配一些额外的空间

这样不会在其他变量上涂写

忘记初始化内存分配：

当你正确调用malloc

但忘记初始化了

程序会遇到未初始化读取 uninitialized read

它会从堆中读取一些未知数据

忘记释放内存

内存泄露memory leak

如果忘记释放内存，在长时间运行的程序来说会有错误

缓慢泄露内存会导致内存不足

如果进程运行时间短，很快就会退出，不用free也是可以的

操作系统会清理其分配的所有的页面

但这是一个环习惯

在用完之前提前释放内存

错误称为悬挂指帧dangling pointer

调用了free但又再次调用malloc来分配其他内容

反复释放内存double free

有可能会导致崩溃

错误调用free（）

free只希望你传入malloc得到的指针

如果传入别的值，坏事会发生

无效释放invalid free

### 14.6

calloc 避免忘记初始化

realloc 用于添加一些东西时

## 15

### 15.0

为了实现高效虚拟化，操作系统尽量让程序自己运行

在关键点时介入interposing，保持对硬件控制

问：

如何高效实现内存虚拟化

如何提供内存所需的灵活性-程序以任何方式访问自己的地址空间

保持控制应用程序可访问的内存位置-只能访问自己的内存空间

硬件地址转换 hardware-based address translation（地址转换address translation，受限直接运行的补充）

硬件对每次内存访问都会处理（如指令获取，数据获取，写入）

将虚拟地址virtual 转换成物理地址physical

只靠硬件时不够的

操作系统要在关键时刻介入 interposing，设置硬件

操作系统必须管理内存manage memory

记录占用和空闲的内存位置

### 15.2

对于程序来说，address space 是从0开始16kb结束

对于虚拟内存来说，操作系统希望把address space放到物理内存的其他位置而不是从0开始

### 15.3

动态重定位dynamic relocation

每个CPU需要两个硬件寄存器：基址base 寄存器和界限bound寄存器--（限制寄存器limit）

这些寄存器可以将address放 space在物理内存的任何位置

同时保证进程只会访问自己的address space

base将虚拟地址转为物理地址

bound确保进程地址在空间范围内 -比如bound设置了16kb，地址找过16就会发生异常

physical address=virtual address+base

（这种重定位是在运行时发生的，我们可以在运行后改变其地址空间- dynamic relocation）

因为两个寄存器在CPU里，这个CPU负责的地址转换叫Memory management unit （MMU）

free list-记录没有使用的物理内存范围

### 15.4

硬件支持：总结

硬件要求 解释

----------------------------------------------------------------------------

特权模式 防止user mode的进程执行特权操作

base bound寄存器 每个CPU需要一堆寄存器来地址转换和界限检查

能转换地址并检查是否越界 用电路完成，这样会非常简单

修改base bound寄存器特权指令 在用户程序运行前，操作系统可以修改这些值

注册异常处理程序的特权指令 操作程序要告诉硬件，如果有异常，就执行哪些代码

能触发异常 如果进程用特权指令或越界内存

-----------------------------------------------------------------------------

我们要两种CPU模式- kernel mode和user mode

只要一bit ，保存在process status mode中，就可以说明CPU的运行模式

每个CPU内存管理单元MMU 需要两个寄存器-base register \bound register

程序运行时，硬件会转换地址并检查地址是否有用

硬件会提供一些特殊指令（特权privileged 指令）来修改这两个寄存器，运行操作系统在切换进程时改变它们

用户程序在尝试非法访问内存时，CPU能产生异常exception

CPU会阻止程序运行，安排系统的exception handler去处理

exception handler会做出正确的回应

### 15.5

操作系统要求 解释

------------------------------------------------------------------------------------

内存管理 为新进程分配内存

从终止的进程回收内存

通过free list来管理内存

base bound 管理 在contact switch时正确设置base\ bound寄存器

异常处理 当发生异常的时候执行的代码

---------------------------------------------------------------------------------------

在进程创建时：

操作系统为进程的地址空间找到内存空间

操作系统会搜索free list找到位置并将其标记为已用

当进程终止时：

操作系统会将此进程的内存放回空闲列表

根据需要清除相关数据结构

操作系统可以改变其他地址空间的物理地址

操作系统会让一个进程停止

将地址空间拷贝到新的位置

更新保存base register，指向新位置

上下文切换时：

操作系统要保存和恢复base 和bound register

当操作系统终止当前进程运行时

会将base和bound register的内存保存在内存中，放在每个进程都有的结构中（进程结构process structure，进程控制块process control block PCB）

当恢复进程时，要设置正确的值

操作系统要提供异常处理程序exception handler

操作系统会在启动时通过特权命令加载这些程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

受限直接访问（动态重定位）

操作系统启动：

（kernel mode）初始化陷阱表

｜

V

（硬件）记住地址：

系统调用处理程序

时钟处理程序

非法内存处理程序

非常规指令处理程序

｜

V

（kernel mode）开始中断时钟

｜

V

（硬件）开始时钟，在xms时中断

｜

V

（kernel mode）初始化进程表，初始化free list

程序系统运行：

（kernel mode）启动进程A

在进程表中分配条目

为进程分配内存

设置base\bound register

从陷阱返回，进入A

｜

V

（硬件）恢复A的寄存器，转向user mode，跳到A的程序计数器

｜

V

（user mode）运行进程A，获取指令

｜

V

（硬件）转换虚拟地址并获取

｜

V

（user mode）执行指令

｜

V

（硬件）如果要显示加载或保存

确保地址不越界

转换虚拟地址并执行

加载\保存

｜（直到时间中断）

V

（硬件）时间中断，转向kernel mode，调到中断处理程序

｜

V

（kernel mode）处理陷阱，调用switch（）例程切换上下文

将寄存器A保存到进程结构A

将进程结构B恢复到寄存器B（都包括base\bound register）

从陷阱返回

｜

V

(硬件)恢复B寄存器，转向user mode，调到B的程序计数器

｜

V

（user mode）运行B，执行错误加载

｜

V

（硬件）加载越界，转向kernel mode ，调到陷阱处理程序

｜

V

（kernel mode）决定终止进程B，回收B的内存，移除B在进程表中的条目

### 15.6

在address space里由于栈区和堆区用到的不多，但使用的物理内存都大，就会有浪费（内部碎片internal fragmentation）

## 16

### 16.0

在堆栈之间有空闲区域internal fragmentation

所以简单的使用base 和 bound register 很浪费

如果剩余的物理地址无法提供连续区域来放置完整的地址空间，进程就无法运行

### 16.1

分段segmentation

在MMU引入不止一对base\bound register

给address space每个逻辑段segment（代码段，栈段，堆段）一对

操作系统能将不同段分到不同的物理区域

### 16.2

硬件在地址转换时候使用段寄存器，如何知道段内的偏移量

显式explicit方式

用虚拟地址的开头几位来显示不同的段（VAX/VMS系统用过）

比如如果是14bit虚拟地址，连两bit就是用来标识段（00代码，01堆，10栈）

硬件会以隐式implicit方法-通过地址产生的方法来确定段

地址有程序计数器产生-代码段

地址基于栈或base指针-栈段

其他为堆段

### 16.3

栈是向上增长，那该怎么办

需要一点硬件支持

硬件需要知道段段增长方向

如 11 1100 0000 0000

11是段，告诉我们这是反向偏移

1100 0000 0000是3KB

假设这段可能是4kb

3kb-4kb=-1kb

偏移量是-1kb

### 16.4

为了节省内存，在地址空间之间共享share是很有用的

额外硬件支持，保护位protection bit

为每个段增加了几个位

标识程序是否能读写该段

如果把该段标位read only

同样的代码就可以被多个进程共享，而且不用担心破环隔离（因为只读）

操作系统秘密的共享内存，进程又不能修改内存，使用假象就可以保持

硬件除了检查是否越界，还要检查特定访问是否允许

### 16.5

目前为止的分段是粗粒度coarse- grained的-地址空间分成较大的段

细粒度fine- grained-将地址空间分为大量较小的段

可以高效利用内存

需要硬件进一步支持

要在内存中保存一种 段标segment table -可以支持创建很多段

### 16.6

物理内存会有许多不同大小的空闲内存-外部碎片external fragmentation

紧凑compact物理内存，重新安排原有的段

操作系统先终止进程，

将数据复制到连续的内存中去

改变他们的段寄存器的值，指向新的物理地址

（这样可以让新的内存请求成功，但拷贝段是密集型的，成本过高）

列表管理算法（试图保留大的内存块用于分配）

相关算法有上千种-最优匹配best fit

## 17

### 17.0

空闲空间管理free-space management

管理的空闲空间由不同大小组成，会变得困难，因为外部碎片external fragmentation

出现在用户级的内存分配库（malloc（），free（））

或操作系统用分段segmentation方法实现虚拟内存

### 17.1

在堆上管理空闲空间的数据结构为free list（包含了管理内存区域中所有空闲块的引用，该数据结构是一个可以追踪空闲空间的数据结构）

一个程序调用malloc（），获得一个指向一块空间的指针，这块区域就属于这个程序了

库不可以移动，直到程序调用free（），所以不可以紧凑compaction来减少碎片

但操作系统实现分段segmentation就可以实现紧凑compaction

### 17.2

分割与合并

free list：

head->add:0,len:10->add:20,len:10->NULL

任何大于10字节的分配请求会失效-没有足够的可连续空间（return NULL）

如果只申请1字节的空间：

分配程序会执行分割splitting

会找到可以满足请求的空间，将其分割

第一块会返回给用户

第二块留在空闲列表中

head->add:0,len:10->add:21,len:9->NULL

合并coalescing：

head->add:10,len:10->add:0,len:10->add:20,len:10->NULL

如果用户请求20字节的空间，会返回失败，因为没有足够的可连续空间

为了避免这个问题，程序会在释放内存时并合并可用的空间

在还一块内存块时

会查看内存块地址和相邻的空间块

如果有相邻，就将它合成较大的空间块

head->add:0,len:30->NULL

追踪已分配空间的大小

大部分分配程序会在header中保存一点额外的信息，它在内存中，通常放在内存块之前

此header包含了一些格外指针加速空间释放

如果用户请求N字节，库不会寻找大小N的空间块，而是寻找N+header块大小的空间块

嵌入空闲列表

如何在空闲内存内部建立一个列表

freelist.c

这个list记录的大小为4080

假设有一个100字节的内存请求

库会找到4080的块然后分割split

一块满足请求，一块是剩余空闲块

假设记录头块header是8字节，那么剩余的是3972（库分配了108字节）

如果要free（）一个空间，小空间块的next指针会指向大空间块的地址

如果不用合并coalesce，没个空闲块的next指针会指向另一个，这会变得很糟

解决：

遍历列表，合并merge相邻块

让堆增长：

sbrk系统调用

会找到空闲的物理内存页

将他们映射到请求进程的address space里

返回堆堆末尾地址

### 17.3

管理空闲空间的基本策略

最优匹配best fit

遍历整个空闲列表

找到大于等于请求大小的空闲块

返回其中最小的一块

（遍历要付出高的性能代价）

最差匹配worest fit

尝试找最大的空闲块

split之后把剩余的放入free list

（遍历，过量的碎片，很高的开销）

首次匹配first fit

找到一个足够大的块，剩余空间留给后续请求

不需要遍历所有空闲块

（有时会让空闲列表开头有很多小块）

下次匹配next fit

多一个指针

指向上次查找结束的位置

避免对列表开头频繁split

### 17.4

其他方式

分离空间列表segregated list

如果某个程序进程申请一种或几种大小的内存

用一个独立的列表来管理这样大小的的对象

其他大小的交给更通用的分配程序

二分伙伴分配程序binary buddy allocator

让合并变得简单

空闲空间首先会被看成2^N大小

当有内存分配时，空闲空间会被递归的一分二（一直除以2）

如果一个8kb空间被释放

程序会检查另一个伙伴（8kb）是否时空闲空间

如果是就可以合并成16kb，然后再检查伙伴（16kb）。。。

查找列表会很慢，缺乏可扩展性scaling

别的算法：

牺牲简单性来换取性能

平衡二叉树，伸展树，偏序树

## 18

### 18.0

分页

把一个address space分成固定大小的单元

每个单元称为一页

把物理内存看成定长槽块的阵列，叫页帧page frame

每个page frame包含一个虚拟内存页

内存会分成许多 页帧page frame

### 18.1

页表page table

为address space 的每个虚拟页面进程地址转换address translation

虚拟页0-》物理page frame 3，virtual page1-〉page frame7

每个进程都有一个这个页表

转换translate

把虚拟地址分成虚拟页面号virtual page number 和偏移量 offset

如果进程的虚拟空间是64字节，每页字节为16

那可以让virtual page number占2位（64/16=4）

让offset占4位（16）

### 18.2

页表非常大

使用将每个进程的页表存在内存中

### 18.3

page table的组织

是一种数据结构

将虚拟页号映射到物理帧号上

PTE是这个数据结构，里面有各种位（如valid bit）还有PFN

### 18.4

分页很慢

### 18.5

内存追踪

## 19

### 19.0

分页会带来较高的性能开销

会额外增加一次内存访问

可以用硬件来加速

地址转换路旁缓冲储存器translation-lookaside buffer（TLB）

也可以叫地址转换缓存address-translation cache

对每次访问，都会检查TLB如果有期望的转换映射，会可以很快完成转换

### 19.1

大致算法

从虚拟地址提取virtual page number

然后检查 translation-lookaside buffer是否有这个VPN

如果有TLB hit，就意味着TLB有该页面的的转换映射

这样就可以从TLB取出page frame number，与原来的offset结合

期望的物理地址就形成了

如果CPU没有找到映射（not hit）

会用转换过的映射来更新translation-lookaside buffer

在地址有效，有相关的访问权限下

### 19.3

处理TLB未命中

搞硬件的不太相信搞操作系统的

所以全部由硬件来处理

以前的硬件有复杂指令集Complex- Instruction set computer（CISC）

硬件必须知道页表在硬件中的任何位置

发生not hit时，硬件会遍历整个page table

取出想要的映射，用它更新translation-lookaside buffer

（x86架构 采用固定多页表multi-level page table）

现代的体系结构都是精简指令级计算机reduced- Instruction set computer

软件管理TLB software- managed TLB

在not hit时硬件会抛出异常

会暂停指令流

将特权升至kernel mode

跳转到陷阱处理程序trap handler（操作程序的一段代码）

然后处理TLB未命中，然后从陷阱返回

程序会再次使用指令导致TLB命中（hit）

与之前的陷阱返回不同

返回后会进行执行操作系统之后的指令

所以在陷入内核时必须保存不同的程序计数器pc

以便将来继续正确的执行

要额外小心TLB未命中递归

### 19.4

硬件TLB内容

有32，64，128项

是全相联的fully associative

映射存在TLB任意位置

硬件会并行查找TLB

内容： VPN｜PFN｜其他位（有效位valid bit-标识该项是不是有效映射，保护位protection-可读可执行可写）

### 19.5

在上下文切换对TLB的处理

TLB包含的映射只对当前进程有效

进行上下文切换时

硬件和操作系统要注意运行的进程不能误读之前进程的地址映射

硬件：

上下文切换直接清空TLB

软件：

把有效valid位设置成0

问题：

每次程序运行，都会触发未命中，开销会高

解决：

会增加硬件支持

实现上下文切换TLB共享

地址空间标识符address space identifier（ASID相当于进程标识符PID，位比PID少）

ASID放在其他位里

上下文切换时，将某个特权寄存器设置为当前的ASID

### 19.6

缓存替换cache replacement

当translation-lookaside buffer插入新项时，替换哪个旧项

替换最近最少使用least- recently-used（LRU）

随机random替换

### 19.7

如果一个程序段时间访问的页数超过了TLB的页数，未命中的次数就会变多，开销会变大

解决的方法就是用更大的页

更大页通常被database management system使用（数据结构大，随机访问）

## 20

### 20.0

系统中，每个进程都有一个页表page table

32位地址空间

4KB的页和4字节的页表项，一个进程有一百万个虚拟页面

如果有一百个进程

就要为内存分配数百兆的内存

问题：

如何让页表page table更小

### 20.1

更大的页

但会导致内存浪费，内部碎片internal fragmentation

### 20.2

结合分页和分段

杂合hybrid

不是为每个address space提供一个page table

而是为每一个段（代码段，堆，栈段）提供一个page table

基址寄存器base

保存该段页表的物理地址

界限bound寄存器

指示页表的结尾

系统中每个进程都有3个与其关联的页表

在上下文切换时，要更改这些寄存器

分段位SN（00为未使用段，01为代码段，10为堆，11位栈）

在TLB未命中时：

硬件会用SN确定时是哪个base bound对

然后将物理地址和VPN结合起来，形成PTE

问题：

页表会变成任意大小

为他们寻找自由空间会很麻烦

### 20.3

多级页表multi-level page table

去掉页表中无效的区域

将页表分成页大小的单元

如果这个页单元的整个页表项PTE无效，就完全不分配该页的页表

这个只是让线性页表的一部分消失

页目录page directory

追踪页表的页项PTE是否有效

会告诉你页表的页在哪里

页目录的每一项叫页目录项 page directory entries

如果一个PDE的valid是1

那他所对应的那个页单元里至少有一个PTE是有效的

问题：

当TLB未命中时

可能需要从内存加载两次才可以命中

（一次为页目录，一次为PTE页表项本身）

比简单的线性页表复杂

### 20.4

反向页表

inverted page table

### 20.5

将页表交换到磁盘

页表仍然有可能太大而无法一次装入内存

一些系统把这样的页表放入 内核虚拟内存kernel virtual memory

在内存压力太大时

将页表中的一部分交换到swap到磁盘

## 21

### 21.0

目前都是假设address space能放入内存

现在我们要支持同时运行巨大的地址空间

目前我们都假设内存的页都是常驻在物理内存中

现在要支持更大的address space

操作系统需要把当前没有在用的那部分地址空间找个地方（要不内存大）储存起来

硬盘hard dick drive和SSD闪存 可以满足这个需求

操作系统可以透明的提供虚拟地址空间假象

### 21.1

交换空间swap space

开辟一部分空间用于移入和移出

将内存的页交换到其中

在需要的时候交换回去

操作系统需要记住给定页的磁盘地址disk address

假设有4页物理内存和8页交换空间和3个进程

3个进程主动共享物理空间

3个进程每一个都有一部分有效页在内存中

剩下的在磁盘的交换空间中

第4个进程所有的页在交换空间中（这个没有运行）

如果运行一个二进制程序（自己的main程序或ls）

代码页最开始在磁盘上

程序运行时会一页一页加载到内存中（现在的方法）

### 21.2

假设有一个硬件管理TLB的系统

如果希望页交换到磁盘，要在原基础上加更多机制

当硬件在PTE查找时，可能发现页不在内存中

这个用新信息来显示-存在位present bit（1为在物理内存中，0在硬盘上）

访问不在物理内存的页，叫页错误page fault

### 21.3

页错误page fault

几乎所有的系统都在软件中处理page fault

问题：

操作系统如何知道所需的页在盘底哪里（可以用PTE的某些位来存地址，项PFN）

当操作系统受到错误时，会在PTE找地址，将请求发送到硬盘，将页读取到内存中

当硬盘IO完成时

操作系统会更新页表，将present bit改成1

更新PTE的PFN来记录新获取页的内存位置

然后再次访问

TLB会说未命中（这次时因为没有更新在TLB中）

### 21.4

如果内存满了

操作系统可以系统换出page out一个或多个页

页交换策略page- replacement policy

选择哪些页来替换replace

### 21.5

页错误处理流程

首先操作系统会先去找free的物理帧FPN

如果没有FPN（内存满了）

会调用页交换算法得到一个空的FPN（算法会先检查是否有空闲页，如果没有，通知swap daemon按需要释放页）

之后要等待IO（从交换空间读取页）

之后把PTE的present bit设为1

把PTE的PFN设为现在的FPN

然后重新在试一下指令（访问数据）

### 21.6

目前为止是只有等到内存满了才会执行replacement的流程（不切实际）

操作系统可以主动预留一些空闲空间

操作系统会设置高水位线high water mark和低水位线low watermark

当操作系统发现有少于LW的页可以用时，后台会让释放内存的线程运行，直到到了HW个可用的物理页

后台线程有时叫交换守护线程swap daemon（守护进程 page daemon）

## 22

### 22.0

替换策略replacement policy

决定在内存压力memory pressure 大的时候把哪一页换出paging out

### 22.1

可以把虚拟内存页视为缓存cache

目标是让缓存未命中cache miss最少（使磁盘取页的次数最少）

让cache hit最多

平均内存访问时间average memory access time

衡量硬件缓存的指标

AMAT=[缓存命中的概率（Phit）\*访问内存的成本（Tm)] + [缓存未命中的概率（Pmiss）\*访问磁盘的成本（Td）]

Phit+Pmiss=1

设有9个虚拟页在内存，1个页在磁盘

那就是9个hit（90%） 1个miss（10%）

设访问内存的成本是100ns

访问磁盘是10ms

AMAT=0.9\*100ns + 0.1\*10ms =90ns+1ms=1ms

如果命中率是99.9%

那AMAT是10.1us快100倍

尽可能避免未命中

### 22.2

最优optimal替换策略（很难实现）

假设访问的页为0 1 2 0 1 3 0 3 1 2 1

在刚开始0 1 2 会未命中，因为缓存开始是空的（冷却未命中cold-start miss）

当到3时缓存满了，用最优策略进行替换

检查0 1 2 未来的访问情况

发现2是在很久以后才会被访问

把2踢出

22.3｜22.4

FIFO 和 random

### 22.5

FIFO 和 random都有可能会提出重要的页（马上会被引用）

频率frequency

如果一个页被访问多次，那不应该替换

近期性recency

近期访问的页，在此访问的几率更大

这些倾向于向循环和数组之类的

最不经常用策略least- frequently-used（LFU）

最少最近用least- recently-used（LRU）

每次访问时

我刚刚访问的放在第一位

别的页将一位

当要替换时，把最后一位的替换掉

在循环时表现不好

### 22.8

实现完美LRU代价比较大

所以用近似LRU

在硬件上面加一个使用位use bit

每当页被读和写时use bit会变成1

但硬件不会把它变成0

这个有操作系统负责

时钟指针clock hand会指向一个最不需要的页

当必须进行页替换时

操作系统会检查当前指向的页是1还是0

如果是1就说明最近用过

然后该页的use bit会变成0

指针会指向下一页

会一直持续这样的操作直到有0的页出现，将其踢出

（如果所有页都是一，那就将所有页都变成0）

### 22.9

考虑脏页

如果这个页已经被修改modified 那它就会变脏dirty

踢出这个页就必须写入磁盘

但干净clean的页（没有修改的）被踢出

就不需要额外的IO

所以时间算法可以修改一下

先把未使用，又干净的页先踢出

无法找到这种页的话，再找脏的未使用页

### 22.10

替换页面不是唯一的策略

页选择策略page selection

操作系统要决定何时将页载入内存

预取prefetching

假设如果代码页p载入内存

代码页p+1也可能载入

如何决定将页面写入磁盘

聚集写入clustering-一次写入大的

一次写入大的比一次写入小的要快

### 22.11

抖动thrashing

正在运行的进程的内存需求超出了可用的物理内存

系统就会不断换页

目前一些系统会采用严格的方式解决内存过载

有些linux会运行 内存不足杀手程序out-of- memory killer

会选择内存密集型进程并杀死它

问题：

如果杀死了某服务器

会导致某些程序不可用

## 23

### 23.1

VMS系统

主要架构师是Dave Cutler（后来开发Windows nt）

VMS是软件创新很好的例子

用于隐藏架构的一些固有的缺陷

### 23.2

VAX-11

为每个进程提供32位虚拟地址空间

每页为512字节

所以虚拟地址为23位VPN

9位offset

由于此系统是分页和分段混合体

VPN前两位是用来显示段的

地址空间的下半部分是进程空间

进程空间的一部分p0有用户程序和堆

p1是栈

地址空间上半部分是系统空间（s）

存操作系统的代码和数据

用于跨进程共享

VMS的页大小非常小

所以首要目标是确保VMS不会用页表占满内存

所有这个系统会为每个进程的每个区域（p0，p1）都提供页表

所以堆栈未使用的空间不会放入表内

内核的虚拟内存中会存放这些表

在分配或增长表时

内核在S段分配自己的内存空间

### 23.3

真实的地址空间

int \*p=NULL;

\*p=10;//硬件在TLB 找VPN miss hit并发现VPN0 被标记为无效，会报错

代码段永远不会在第0页开始（该页标记为不可访问，为检测空指针null pointer提供支持）

内核虚拟地址空间是每个地址空间的一部分

在上下文切换时操作系统会改变p0 p1指向即将运行的进程的页表

但不会改变S的base bound register

内核映射到每个地址空间的原因：

如果操作系统收到用户提交的指针

很容易将数据复制到它自己的结构

操作系统不希望用户程序可以读取或写入操作系统的数据和代码

因此操作系统要支持页面的不同保护级别才可以用该功能

系统的数据和代码会被设置成比用户的更高的保护级别

### 23.4

分段FIFO

每个进程都有一个可以保存在内存中的最大页数 驻留集大小Resident set size RSS

当一个进程的页超过RSS

先入的页会被踢出

二次机会列表second-chance list

页在被踢出前会放入其中

如当进程P超过RSS时

会将每个进程FIFO中移除一个页

干净页会放在全局干净列表的尾部

脏页会放在全局脏列表的尾部

如果进程P遇到页错误

就可以在这两个列表中找回自己的页（这样就避免了磁盘IO）

如果进程Q需要空闲页

就在干净列表里抽出第一个页

页聚集（clustering）

磁盘在大小传输中效果更好

VMS会将全局脏列表的页聚集在一起

然后一举写入磁盘

### 23.5

写时复制copy on write

操作系统需要将一个页面从一个地址空间复制到另一个地址空间

可以将其映射带地址空间而不是复制

然后在两个地址空间标记为只读

这样可以提升速度

如果一个地址空间想写入页面

就会陷入操作系统

会分配一个新页，填充数据

将这个新页映射到要写入的那个地址空间

# Concurrency

## 26

### 26.0

线程thread

一个程序只有一个执行点（一个程序计数器PC）

多线程multi- threaded有多个执行点（多个程序计数器PC）

他们虽然每个像一个进程

但他们共享地址空间

可以访问相同的数据

每个线程有自己的PC

当它们运行在同一个处理器时

从一个线程切换到另一个线程

会发生上下文切换context switch

我们需要一个或多个线程控制块thread control block（tcb）

在切换时不需要改变address space（不需要切换页表）

（如果是进程，要将状态保存在进程控制块process control block（pcb））

在多线程中

在address space中，每个线程都有一个栈

所有位于栈上的变量，参数，返回值等将被放在称为线程本地thread-local的地方（就是相关线程的栈）

### 26.3

假设线程1进入代码区域

要增加一个计数器

它将counter多值（50）加载到寄存器eax

eax=50（线程1）

然后向寄存器+1

eax=51

现在，时钟中断发生了

操作系统将当前运行的线程的状态（PC，寄存器，eax）保存到thread control block（TCB）

线程2被选中

会将counter的值放入它的eax（每个线程都有属于自己的寄存器，context switch时，会让代码将寄存器虚拟化virtualised）

counter的值还是50（因为线程1没有保存counter）

然后将eax+1

然后把内容保存到counter

然后又发生context switch

线程1恢复运行

其eax里面还是51

最后保存这个51到counter

（这些结果为不确定indeterminate计算）

我们将此段代码称为临界区critical section

我们想要的互斥mutual exclusion-保证一个线程在临界区运行其他线程被阻止进入临界区

### 26.4

解决方法

希望以原子的方式atomically执行（一般做不到）

同步原语synchronization primitive

使用硬件同步原语

加上操作系统的一些帮助

临界区critical section-是访问共享资源的一段代码，资源通常是变量或者数据结构

竞争条件race condition-出现在当多个线程大致同时进入critical section

不确定性indeterminate-程序由一个或多个race condition组成，输出取决于哪些线程何时运行，结果不一

互斥mutual exclusion-保证只有一个线程进入critical section

## 27

### 27.3

锁lock

如果在调用pthread\_mutex\_lock()时没有其他线程持有锁

该线程将持有锁然后进入该线程

当另一个线程调用pthread\_mutex\_lock()

该函数不会返回直到第一个线程调用pthread\_mutex\_unlock()把锁释放

问题：

缺乏正确的初始化lack of proper initialization

pthread\_mutex\_t lock=PTHREAD\_MUTEX\_INITIALIZER

将锁设置成默认值

初始化的动态方法

int rc=pthread\_mutex\_t(&lock,NULL);

assert(rc==0);

第一个参数是锁本身地址

（通常使用这个）

当用完锁时

调用pthread\_mutex\_destory()

pthread\_mutex\_trylock

如果锁被占用

这个将会失败

pthread\_mutex\_timedlock

会在超时或获取锁后返回

两个都是获取锁

### 27.4

条件变量condition variable

pthread\_cond\_t cond=PTHREAD\_COND\_INITIALIZER

(可以用pthread\_cond\_init()来初始化和pthread\_cond\_destroy())

一个线程在等待另一个线程执行某些工作，这很有用

pthread\_cond\_wait()

使调用的线程进入休眠状态

等待其他线程发出信号

当程序某些内容发生变化时，休眠的线程可能会被影响

第二个参数是lock

因为在让线程进入休眠外

会让调用者睡眠时释放锁

在被唤醒时，会重新获取锁

pthread\_cond\_signal()

在用条件变量时要带有lock

## 28

### 28.1

锁lock

lock是一个锁变量lock variable

保存了锁在某一时刻的状态

要么是可用的-没有线程持有lock

要么是被占用的-有一个线程持有lock

（也会保存一些隐藏信息，如持有锁的线程，讲求锁的线程队列）

lock（）

尝试获取lock

如果没有其他线程持有lock

该线程会获得lock并进入critical section

该线程会被称为拥有者owner

如果另外一个线程调用lock（）

但现在已经有线程持有lock了

所以调用lock（）时是不会返回的

unlock（）

owner 一旦调用unlock（）

lock就变为可用的了（在没有其他线程调用lock（）的情况下）

如果有线程等待（卡在lock（））

它会注意锁的变化并获取锁，进入critical section

### 28.2

pthread锁

POSIX库把锁称为互斥量mutex（被用来与线程之间的互斥）

POSIX的lock（）和unlock（）会传入一个变量（lock的地址）

使用不同的lock来保护不同数据或结构

可以增加并发

### 28.4

评价锁

建立标准

1，是否可以完成基本任务-互斥mutex

2，公平性fairness，当锁可用时，是否每一个竞争的公平线程都能抢到锁（是否会有竞争锁会饿死starve）

3，性能performance，使用锁的时候增加的时间开销，（在没有竞争的情况和CPU上多个线程竞争情况上要考虑）

### 28.5

控制中断

在critical section关闭中断

因为没有中断，线程可用确定代码会继续执行下去，从而实现原子化

缺点：

此方法会让所有线程拥有特权操作

不支持多处理器，多线程运行在不同的CPU上，每个线程会进入同一个critical section，关闭中断不会有用

会导致中断流失，会有系统问题

效率低

### 28.6

让硬件支持锁

测试并设置指令test-and-set instruction（也叫原子交换 automatic exchange）

如果用代码的话（软件）：

正确性：

有很大概率两个线程会进入critical section

性能：

在线程已经有所时

用的是自选等待spin-waiting（像while，不停检查值）

浪费时间

### 28.7

如果不用硬件的话是不行的

一些系统提供了这个指令，支持这种lock（test-and-set指令）

SPIC-ldstub（load/store unsigned byte）

x86-xchg（atomic exchange）

int TestAndSet(int \* old, int new){

int getOld=\*old;

\*old=new;

return getOld;

}//这个代码会原子地atomically执行

//既可以测试旧值也也可以设置新值

### 28.7

评价自旋锁

有正确性correctness

没有公平性fairness

无法保证，可能会永远自旋

性能performance

自旋锁性能开销相当大

在单CPU下

自旋一个时间片

浪费CPU周期

多CPU没问题

28，9

比较并交换

SPARC是compare-and-swap

x86是compare-and- exchange

int compare\_And\_swap(int \*ptr, int expect, int new){

int actual=\*ptr;

if(actual==expect){

\*ptr=new;

}

return actual;

}//减少赋值

### 28.10

链接加载load-linked和条件储存store- condition

load-linked

和加载指令类似

store- condition

只有上一次加载的地址没有被更新时（调用load-linked）

才会被更新并返回1（超过）

不然返回0（失败）

while（LoadLinked(&lock->flag) || !StoreConditional(&lock->flag,1));

### 28.11

获取并增加fetch-and-add

可以atomically返回地址的旧值

并让该值增加1

ticket锁

一个数据结构

带有turn

和ticket

当调用lock（）时

把ticket（0）作为参数放入fetch and add（ticket变为1）

返回值（0）会和turn（0）比较

当第二个线程调用lock（）时

把ticket（1）作为参数放入fetch and add（ticket变为2）

返回值（1）会和turn（0）比较

回旋锁

以此类推

当调用unlock（）时

把turn（1）作为参数放入fetch and add（turn 变为1，这样就可以让第二个线程进入critical section）

这样可以让所有线程抢到锁

### 28.12

自旋效率低下

一个线程会自旋检查一个不会改变的值

浪费整个时间片

如果有N个线程竞争一个锁

情况会更糟糕

所以只有硬件支持是不够的

还需要操作系统支持

### 28.13

如果critical section的线程context switch，别的线程只会自旋

方法1:

在自旋的时候，放弃CPU

yield():线程会调用它主动放弃CPU，让其他线程运行

该系统调用可以让running态变为ready态

问题：

如果有100个线程

只有一个线程持有锁

其余99个线程会一直处于让出

成本还是很高

而且持有锁的线程很可能会饿死

### 28.14

需要一个队列来保存等待锁的线程

park（）可以让调用线程休眠

unpark（threadID）会唤醒线程

在获取不到锁时休眠，在有锁时唤醒

1 typedef struct \_\_lock\_t {

2 int flag;

3 int guard;

4 queue\_t \*q;

5 } lock\_t;

6

7 void lock\_init(lock\_t \*m) {

8 m->flag = 0;

9 m->guard = 0;

10 queue\_init(m->q);

11 }

12

13 void lock(lock\_t \*m) {

14 while (TestAndSet(&m->guard, 1) == 1)

15 ; //acquire guard lock by spinning

16 if (m->flag == 0) {

17 m->flag = 1; // lock is acquired

18 m->guard = 0;

19 } else {

20 queue\_add(m->q, gettid());

21 m->guard = 0;

22 park();

23 }

24 }

25

26 void unlock(lock\_t \*m) {

27 while (TestAndSet(&m->guard, 1) == 1)

28 ; //acquire guard lock by spinning

29 if (queue\_empty(m->q))

30 m->flag = 0; // let go of lock; no one wants it

31 else

32 unpark(queue\_remove(m->q)); // hold lock (for next thread!)

33 m->guard = 0;

34 }

问题：

在调用park（）之前（已经加入了队列）突然开始上下文切换

正好这个持有锁的线程调用unlock（）

然后调用了unpark（）

移除了刚刚加入队列的threadID

这样那个线程就会永久休眠

解决：

用setpark（）

如果已经调用了unpark（）

在调用setpark（）

就会立即返回，而不是一直休眠

### 28.15

Linux提供了futex（类似于park）

futex\_wait(address,expected)

如果address处的值==expected

就会休眠

否则就会返回

futex\_wait(address)

唤醒等待队列中的一个线程

### 28.16

二阶段锁two- phase lock

在很快要放弃锁的时候

自旋spin其实很有用

第一阶段会先spin一段时间，希望能获取锁

如果第一阶段没有获取，到第二阶段，它会休眠

(杂合hybrid)

## 29

### 29.0

如何给数据结构加锁

### 29.1

简单无扩展计数器

普通的线程安全thread safe

在线程越多的时候性能越差，耗时越多

多处理器上的多线程运行像单线程一样快（完美扩展perfect scaling）

可扩展计数器

可扩展计数器很重要

懒惰计数器用多个局部计数器和一个全局计数器

先增加它的局部计数器

过一段时间之后用锁加到全局计数器

让全局计数器变为零

这个取决于一个值S（sloppiness）-到这个值S之后就会加到Global里

s越大，扩展性越大

### 29.2

并发链表

。。。

扩展列表

手锁hand-over-hand locking

每个链表里都有一个锁，代替之前到一个锁

遍历列表时

抢到下个节点的锁

释放当前节点的锁

（开销巨大）

### 29.3

并发队列

一个叫队列的数据结构

里面包含链表的头和尾

有两个锁-一个负责头，一个负责尾

增加时尾部增加

出队列从头部出

### 29.4

散列表

2 typedef struct \_\_node\_t {

3 int key;

4 struct \_\_node\_t \*next;

5 } node\_t;

6

7 // basic list structure (one used per list)

8 typedef struct \_\_list\_t {

9 node\_t \*head;

10 pthread\_mutex\_t lock;

11 } list\_t;

12

13 void List\_Init(list\_t \*L) {

14 L->head = NULL;

15 pthread\_mutex\_init(&L->lock, NULL);

16 }

17

18 int List\_Insert(list\_t \*L, int key) {

19

20 node\_t \*new = malloc(sizeof(node\_t));

21 if (new == NULL) {

22 perror("malloc");

23

24 return -1; // fail

25 }

26 new->key = key;

pthread\_mutex\_lock(&L->lock);

27 new->next = L->head;

28 L->head = new;

29 pthread\_mutex\_unlock(&L->lock);

30 return 0; // success

31 }

32

33 int List\_Lookup(list\_t \*L, int key) {

34 pthread\_mutex\_lock(&L->lock);

35 node\_t \*curr = L->head;

36 while (curr) {

37 if (curr->key == key) {

38 pthread\_mutex\_unlock(&L->lock);

39 return 0; // success

40 }

41 curr = curr->next;

42 }

43 pthread\_mutex\_unlock(&L->lock);

44 return -1; // failure

45 }

#####################################

1 #define BUCKETS (101)

2

3 typedef struct \_\_hash\_t {

4 list\_t lists[BUCKETS];

5 } hash\_t;

6

7 void Hash\_Init(hash\_t \*H) {

8 int i;

9 for (i = 0; i < BUCKETS; i++) {

10 List\_Init(&H->lists[i]);

11 }

12 }

13

14 int Hash\_Insert(hash\_t \*H, int key) {

15 int bucket = key % BUCKETS;

16 return List\_Insert(&H->lists[bucket], key);

17 }

18

19 int Hash\_Lookup(hash\_t \*H, int key) {

20 int bucket = key % BUCKETS;

21 return List\_Lookup(&H->lists[bucket], key);

22 }

有100个散列桶

利用空间换时间

有很多分支

重要读取会很快

性能高

## 30

### 30.0

锁并不是并发程序设计的唯一原语

线程会在某一条件condition满足后会继续运行

join（）等待是如何实现的呢？

### 30.1

线程可以使用条件变量condition variable

condition variable 是一个显式队列

当某些状态不满足的话

线程会把自己加入队列

waiting该条件

pthread\_cond\_t c;

线程要睡时用wait（）

线程要唤醒时用signal（）

（hold the lock when calling signal）

### 30.2

生产者/消费者（producer/consumer）

界缓冲区bounded buffer

producer 有一个或多个

它们把生成的数据放入缓冲区

consumer 有一个

它从缓存区取走数据

（如producer把http请求放入队列（缓冲区），consumer取走请求并处理）

（管道也是用bounded buffer- cowsay helloCaesar|cowsay 左边是producer右边是consumer）

唤醒只会唤醒一个线程

### 30.3

覆盖条件covering condition

pthread\_cond\_broadcast()唤醒所有等待的线程（可能会影响性能）

## 31

### 31.1

信号量

初始化：

#include <semaphore>

sem\_t s;

sem\_init(&s,0,1);(次函数在mac上被弃用，所以可能会没有用)

第三个参数将值初始化为1

第二个参数表示信号量在同一个进程的多个线程是共享的

用法：

int sem\_wait()

要么在信号值>=1是直接返回

要么让调用线程挂起（自己睡眠，放弃处理器）

知道有post（）操作

int sem\_post()

直接增加信号量的值

如果有等待的线程

只唤醒其中的一个

当信号量为负数时

值为等待线程的个数

### 31.2

将信号量作为锁

锁只有两个状态-持有和未持有

二值信号量binary semaphore

调度程序状态schedular state：

运行

就绪

等待别的线程运行

睡眠

### 31.3

生产者producer 消费者consumer

1 int buffer[MAX];

2 int fill = 0;

3 int use = 0;

4

5 void put(int value) {

6 buffer[fill] = value; // line f1

7 fill = (fill + 1) % MAX; // line f2

8 }

9

10 int get() {

11 int tmp = buffer[use]; // line g1

12 use = (use + 1) % MAX; // line g2

13 return tmp;

14 }

Figure 31.9: The Put And Get Routines

1 sem\_t empty;

2 sem\_t full;

sem\_t lock;

3

4 void \*producer(void \*arg) {

5 int i;

6 for (i = 0; i < loops; i++) {

7 sem\_wait(&empty); // line P1

sem\_wait(&lock);

8 put(i); // line P2

sem\_post(&lock);

9 sem\_post(&full); // line P3

10 }

11 }

12

13 void \*consumer(void \*arg) {

14 int i, tmp = 0;

15 while (tmp != -1) {

16 sem\_wait(&full); // line C1

sem\_wait(&lock);

17 tmp = get(); // line C2

sem\_post(&lock);

18 sem\_post(&empty); // line C3

19 printf("%d\n", tmp);

20 }

21 }

22

23 int main(int argc, char \*argv[]) {

24 // ...

sem\_init(&lock,0,1)//二值信号锁-防止在MAX>1时多个线程同时进入put（）

25 sem\_init(&empty, 0, MAX); // MAX buffers are empty to begin with...

26 sem\_init(&full, 0, 0); // ... and 0 are full

27 // ...

28 }

Figure 31.10: Adding The Full And Empty Conditions

### 31.5

读者-写者 锁 reader- writer lock

rwlock\_acquire\_writelock()获得锁

rwlock\_release\_writelock()释放锁

通过writelock信号量保证只有一个写者能获得锁进入critical section

1 typedef struct \_rwlock\_t {

2 sem\_t lock; // binary semaphore (basic lock)

3 sem\_t writelock; // used to allow ONE writer or MANY readers

4 int readers; // count of readers reading in critical section

5 } rwlock\_t;

6

7 void rwlock\_init(rwlock\_t \*rw) {

8 rw->readers = 0;

9 sem\_init(&rw->lock, 0, 1);

10 sem\_init(&rw->writelock, 0, 1);

11 }

12

13 void rwlock\_acquire\_readlock(rwlock\_t \*rw) {

14 sem\_wait(&rw->lock);

15 rw->readers++;

16 if (rw->readers == 1)

17 sem\_wait(&rw->writelock); // first reader acquires writelock

18 sem\_post(&rw->lock);

19 }

20

21 void rwlock\_release\_readlock(rwlock\_t \*rw) {

22 sem\_wait(&rw->lock);

23 rw->readers--;

24 if (rw->readers == 0)

25 sem\_post(&rw->writelock); // last reader releases writelock

26 sem\_post(&rw->lock);

27 }

28

29 void rwlock\_acquire\_writelock(rwlock\_t \*rw) {

30 sem\_wait(&rw->writelock);

31 }

32

33 void rwlock\_release\_writelock(rwlock\_t \*rw) {

34 sem\_post(&rw->writelock);

35 }

问题：

读者可能会饿死写者（读者太多）

### 31.6

sem3.c

### 31.7

实现信号量

## 32

### 32.2

非死锁缺陷

违反原子性atomicity violation

1 Thread 1::

2 if (thd->proc\_info) {

3 ...

4 fputs(thd->proc\_info, ...);

5 ...

6 }

7

8 Thread 2::

9 thd->proc\_info = NULL;

解决方法：加锁

错误顺序order violation

1 Thread 1::

2 void init() {

3 ...

4 mThread = PR\_CreateThread(mMain, ...);

5 ...

6 }

7

8 Thread 2::

9 void mMain(...) {

10 ...

11 mState = mThread->State;

12 ...

13 }

如果线程1没有先执行而去先执行线程2

那mThread指向的是空

就会报错

解决方法：加锁和条件变量，在线程1没有执行前如果执行线程2就将它睡眠

大部分非死锁就是atomicity violation和order violation

### 32.3

死锁缺陷

如：

线程1有lock1，在等待lock2

线程2有lock2，在等待lock1

如何发生：

在大型代码库里

组件之间有很多依赖

例子：

虚拟内存需要访问文件系统将数据从磁盘读到内存页

文件系统要和虚拟内存交互，去申请一页内存

用这一页存放读到的块

预防：

循环等待

让代码不断产生循环等待

total ordering-每次都是先申请L1 在申请L2

partial ordering-在代码开头注释 加锁的顺序

持有并等待

假如要获取锁L1，L2

就在把获取锁的部分设为critical section

用另外一个锁L\_p来进行atomically

不适用于封装

非抢占

1 top:

2 lock(L1);

3 if (trylock(L2) == -1) {

4 unlock(L1);

5 goto top;

6 }

trylock（）会尝试获取锁，失败会返回-1

问题：

如果抢到L1的话

代码分配了一些内存

如果L2没抢到

又要释放内存（浪费）

互斥

通过调度避免avoidance

假设有两个CPU

有4个线程

线程1有锁1，2

线程2有锁1，2

线程3有锁1

线程4没有锁

那就把有相同锁的线程1，2交给同一个CPU

就不会同时运行

（会限制开发）

检查恢复

允许死锁偶尔发生

检查到死锁时再做相应行动

## 33

### 33.0

基于事件的并发event-based concurrency

不用线程，如何构建并发服务器

### 33.1

事件循环event loop

循环获得事件并去处理

处理时间的程序event hander

处理事件时是系统的唯一活动

调度就是决定接下来处理哪个事件

### 33.2

select（）（pull（））

检查是否有关注的进入IO

如果检查是否有网络数据包到达

传入readfds，writefds，errorfds

检查某些描述符是否已经准备好读取，写入或异常处理

问题：

基于事件的方法只允许主事件循环

没有多线程

如果有IO操作（read open）

就会浪费资源

### 33.6

异步IO

asynchronous IO

aio control block

struct aiocb

int 文件描述符

off\_t 文件内的偏移量

size\_t 长度的请求

void \* 目标内存位置

用异步调用来读取文件

int aio\_read（struct aiocb )

如果成功，会立刻返回

用来检查aiocbp请求是否完成

int aio\_error(aiocb )

如果没有，返回EINPROGRESS

但是一直循环检查是否完成是不符合实际的

用中断方法interrupt

用信号在异步IO完成时通知应用程序

# Persistence

## 36

### 36.1

系统架构

CPU用memory bus 连接到内存

一些高性能IO设备（显卡）通过常规IO bus链接到系统

外围总线peripheral bus如USB它们将最慢的设备连接到系统（磁盘，鼠标）

越快的总线越短

让高性能设备离CPU近一点

低性能离CPU远一点

### 36.2

标准设备

包含两个部分重要组件：

1.向系统其他部分展现硬件接口interface

2.内部结构internal structure

复杂的会包含CPU，memory，和特定芯片

### 36.3

一个简化设备接口包含3个寄存器：

状态status寄存器

读取查看设备当前状态

命令command寄存器

通知设备执行某具体任务

数据data寄存器

将数据传输给设备或从设备接收数据

标准协议

1.操作系统会反复读取状态寄存器，直到变为就绪状态（polling 设备-问它在做什么）

2.操作系统下发数据到寄存器

将磁盘块传递给设备

如果CPU参与数据移动

称为programed IO

3.将命令写入命令寄存器

这样设备就知道已经准备好了

开始执行

4.polling 设备， 判断是否完成任务

### 36.4

利用中断interrupt减少CPU开销

向设备发出请求

让其对应进程休眠

切换执行其他任务

当完成了以上操作，会抛出硬件中断

先定义好中断服务例程interrupt service routine（ISR）

一小段系统代码

求结束之前进程的请求

并唤醒等待IO的进程继续执行

### 36.5

中断允许用IO的时候重叠overlap

如果不用中断

进程在IO的过程中只会自旋spin

用了interrupt

可以在用IO的时候让其他进程获得CPU控制权

中断不是总是好的

如果一个设备性能很高，处理快

CPU一次轮询就可以返回结果

用中断反而会让系统变慢

解决：

可以用混合策略hybrid

先轮询polling一段时间

再interrupt

在网络时最好不要用中断：

网路接收大量数据包

如果每接收一个数据包用一次中断

那可能导致系统发生livelock-无法处理用户层请求

解决：

用合并coalescing

在中断抛出之前等待一小段时间

在此期间，其他请求可能很快完成

多次中断就可以合为一次中断

### 36.5

在用磁盘IO时

会先把数据拷贝到磁盘

这段时间会浪费时间和算力

使用DMA direct memory access

操作系统会通过编程告诉DMA“引擎数据在内存的位置，要拷贝的大小，要拷贝到哪个设备”

这样操作系统就可以处理其他请求了

### 36.6

硬件如何与设备通信

方式1

用明确的IO指令

当需要发生数据给设备时

调用者指定一个存入数据的特定寄存器和一个代表设备的特定端口-特权指令privileged

操作系统是唯一一个可以与设备交互的实体

方法2:

用内存映射IO memory mapped IO

硬件将设备寄存器作为内存地址memory address提供

当访问设备寄存器时

操作系统会（读取或写入）到内存地址

硬件将把读取或写入到内存地址的数据转移到设备上，而不是物理内存

### 36.7

每个设备都有非常具体的接口，如何将它们纳入操作系统

用抽象abstraction技术

在最底层，操作系统一部分软件知道设备如何工作（驱动程序device driver）

### 36.8

与设备交互简单协议

等待驱动就绪-读取状态寄存器，直到驱动Ready而非忙碌

向命令寄存器写入参数-如写入扇区数，逻辑块地址LBA，驱动编号

开启IO-发送读写命令到命令寄存器

数据传送（针对写请求）-等待状态为ready和DRQ（驱动请求数据），向数据端口写入数据

中断处理-在传送结束后出发中断处理程序

错误处理-在结束后检查状态寄存器，如果error，可以读取错误寄存器来或取信息

## 37

### 37.0

磁盘驱动器hard disk drive

持久数据储存的主要形式

### 37.1

hard disk drive接口

drive 有大量扇区（512字节）组成

磁盘有N个扇区

扇区从0～N-1编号

0～N-1是drive的地址空间address space

单个扇区（512字节）的写入是atomic（完整地完成）

支持多扇区操作

### 37.2

盘片platter

是一个圆形坚硬的表面

通过引入磁性变化来永久储存数据

磁盘可能有一个或多个platter

每个platter有两面-每面称为表面

由铝制成涂上磁性层

即使驱动器断电也能持久储存数据未

主轴spindle

所有platter围绕spindle

spindle 连接到一个电机

以一个恒定的速度旋转盘片

旋转速率 rotation per minute RPM——典型数值7200～15000RPM

数据在表面上被编码，一个圆称为磁道

一个表面包含特别多track

读写操作由磁头disk head完成

每个表面都有一个disk head

所有磁头连在一个disk arm上

### 37.3

单磁道延迟：旋转延迟

简单磁盘中

等待期望的扇区旋转到磁头下

这种等待叫旋转延迟rotational delay

多磁道：寻道空间

现在的磁盘有数以百万计track

驱动器要先将disk arm移动到正确的磁道-寻道seek过程

seek有许多阶段：

disk arm移动的加速加速阶段

disk arm的惯性滑动

disk arm的减速

disk head小心地放下-停放时间settling time（在0.5～2ms）

当正确的扇区到达disk head时

传输transfer

数据从表面读取或传输到表面

其他细节：

需要驱动器采用磁盘偏斜track skew——确保即使在跨越track边界时，顺序读取也可以方便服务

外圈track 比内圈的更多-称为多区域multi- zoned

磁盘被组成多个区域

每个区域里一个track有相同数量多扇区

track buffer

该buffer是少量内存（8～16MB）

驱动器可以用这个内存保存从磁盘读取或写入多数据

可以让驱动器快速响应所有后续请求

### 37.4

IO时间=Tseek + T旋转 + T transfer

容量：300GB

平均seek时间：4ms

RPM：15000

最大传输速度：125MB/S

磁盘：4

缓存：16MB

Tseek=4ms，T旋转=（1/（15000/60））/2=每次旋转要2ms T传输=30ups

随机random工作负载

向磁盘随机位置发送小的读取请求（4kb）

T io=6ms

假设读取4KB

那速率是0.66MB/s

顺序sequential工作负载

从磁盘连续读取大量扇区，不会跳过

T io=800ms

速率为125MB/s

### 37.5

磁盘调度

最短寻道时间优先SSTF

（shortest-seek-time- first）

选择在最近track上的请求先完成

SSTF不是万能的

饥饿starnation

无法利用驱动器的几何结构-可以用最近块优先解决NBF nearest block first

电梯elevator 算法

SCAN

如果请求的块在这次扫一遍中已经服务过了

不会立即处理

排队等待下次扫一遍

circular-scan

从外圈扫到内圈

从内圈扫到外圈

最短定位时间优先 SPTF shortest positioning time first

如果seek的时间比旋转要快得多

这个要比SSTF要好

## 38

### 38.0

廉价冗余磁盘阵列(RAID)redundant array of inexpensive disks

使用多个磁盘一起构建

更快更大更可靠

由多个磁盘，内存和一个和多个处理器来管理系统

硬件RAID像一个计算机系统来管理一组磁盘

透明transparent-对于主机看起来像一个大磁盘，所以可以简单的用RAID替换该磁盘

提高了可部署性- deployability

### 38.1

接口和RAID内部

当向RAID发出IO请求时

会先计算要访问的磁盘的数量

然后发出一个或多个物理IO来执行操作

RAID构建为单独的硬件盒

通过标准连接（SCSI，SATA）接入主机

它包括微控制器，运行固件，和指导RAID的操作

也包括DRAM储存器-在读取和写入时用来缓存

### 38.2

故障模型

检测从某些类型的故障中恢复

故障-停止fail-stop故障模型

工作状态

所有块都可以读取或写入

故障状态

块会永久消失

### 38.3

评估RAID-3个方面

容量capacity

RAID客户端可用的容量有多少

可靠性reliability

给定设计允许有多少磁盘故障

性能performance

取决于磁盘阵列的工作负载

### 38.4

RAID 0级：条带化stripe

磁盘0 【0，4，8】

磁盘1 【1，5，9】

磁盘2 【2，6，10】

磁盘3 【3，7，11】

以旋转的方式将磁盘阵列分布在磁盘上

对数组的连续块请求时

从阵列中获取最大的并行性

0，1，2，3在同一条带上

目前是只有一个块放在下一个磁盘上

也可以用两个块

磁盘0 【0，1，4，5】

磁盘1 【2，3，6，7】

块的大小主要影响整列性能

如果块较小

许多文件要跨越多个磁盘进行stripe

增加了对单个文件读写的并行性

依靠多请求来增加吞吐量

较大的块

可以减少定位时间

大多数块的大小为64KB

容量capacity

顶级

可靠性reliability

不行

任何磁盘发送故障都会导致数据丢失

性能performance

并行

使所有磁盘为用户IO服务

单请求延迟（用来揭秘单个请求存在多少并行性）

设：平均寻道时间7ms，平均旋转延迟3ms，磁盘传输速度50MB/s

吞吐量为N（磁盘数量）

S（当磁盘顺序带宽）

R（当磁盘随机带宽）

顺序sequential（如一个访问1MB请求）

10MB连续传输

7ms+3ms+（10/50）s=210ms

Rio=10MB/210ms=47.6MB/s

N\*S

随机random（每个请求很小，而且位置随机）

10KB随机传输

7ms+3ms+（10/50）ms=10.195ms

Rio=10KB/10.195ms=0.981MB/s

N\*R

### 38.5

RAID 1级：镜像

在stripe的基础上

在每个块上多一个副本

每个副本在不同的disk上

磁盘0 【0，1，4，5】

磁盘1 【0，1，4，5】

磁盘2 【2，3，6，7】

磁盘3 【2，3，6，7】

可以在任意副本上读取（如读取逻辑块6，可以在磁盘2，3读取）

容量capacity

价格昂贵减少了一半

可靠性reliability

表现良好

允许任何一个磁盘发送故障

性能performance

与0级相同

写入需要二次写入（但是是并行，大致等于单词写入）

但是逻辑写入要在物理写入之后

要遭到最差的seek和旋转延迟

吞吐量为N（磁盘数量）

S（当磁盘顺序带宽）

R（当磁盘随机带宽）

顺序写入的数据只能分配一半的disk，另一半用来做副本

顺序读取：读取时，一个磁盘会收到其他块的请求，跳过块的旋转时，不会提供有用的带宽

N/2 \* S

随机读取的话可用用来分配读取

N\*R

随机写入的话，另一半用来做副本

N/2 \*R

### 38.6

RAID 4级：通过奇偶校验节省空间

奇偶校验parity

可以用XOR

基数个1为1

偶数个1为0

所有行的1必须是偶数

用parity从故障中恢复

磁盘0 【00，11，01，00】

磁盘1 【10，11，00，11】

磁盘p 【10，00，01，11】

容量capacity

允许一个磁盘作为parity

N-1

可靠性reliability

容许一个磁盘故障，可以恢复数据

性能performance

（N-1）\* S

优化：全条写入full-stripe write

当在有一条要写入的话

可以先计算奇偶校验位然后并行写入

随机读取：

R\*（N-1）

随机写入：

p也要更新（如何正确更新）

加法奇偶校验additive parity

在写入时

并行读取其他块进行计算

减法奇偶校验subtractive parity

通过比较旧数据和新数据

来确定是否更改校验位

C（old） xor C（new）xor P（old）

需要执行两次IO一次读取一次写入

R/2（非常糟糕）

### 38.7

RAID 5级-旋转奇偶校验

工作原理与RAID-4几乎完全相同

只是将奇偶校验计数器旋转

磁盘0 【0，3，p2，7】

磁盘1 【1，p1，4，p3】

磁盘2 【p0，2，5，6】

消除4级的奇偶校验磁盘贫瘠

并发修改校验位

（R）\*（N/4）

## 39

### 39.1

文件file：

是一个线性字节数组

每个字节都可以读取和写入

文件都有某种低级的名称low level name-通常称为inode name

文件系统仅仅将数据永久存在磁盘上

目录directory（文件夹）：

一个目录，像文件

也有low level name

内容非常具体

包含一个对的列表

假设inode为10

用户可读名为“foo”

”foo“所在的目录会有一个对（“foo”，10）

用可读名称映射到低级名称

目录树directory tree

该树下储存所有文件

根目录（“/”）

/foo/foo/bar.txt

bar.txt-第一个是任意名称，第二个是文件类型type

### 39.3

创建文件

用open系统调用完成

调用open（）传入O\_CREAT标志，程序会创建一个新文件

O\_CREAT:创建文件

O\_WRONLY：只写

O\_TRUNC:如果文件已经存在，删除所有内容

以前的创建方式为create()

open会返回一个值：文件描述符

文件描述符file descriptor

是一个整数

每个进程私有的

用于访问文件

可以看作文件类型的对象指针

可以调用它的方法（read write）

### 39.4

读取和写入

一般第一个文件描述符都是3

012为标准输入-进程可以读取

标准输出-进程可以写入已将信息显示到屏幕

标准错误-可以写入错误消息

### 39.6

fsync(int fd)

在调用write时

只会告诉系统要将数据持久存入

系统可能会进行缓冲而不是直接开始写入

当调用该接口时

文件系统会将所有dirty数据强制写入磁盘

一旦所有数据写入完成，fsync（）就会返回

### 39.7

文件重命名

mv foo bar

rename（char\*old，char\*new）系统调用

这是atomic 调用（不会出现中间状态）

### 39.8

元数据metadata

每个文件的大量信息（大小，inode号）

查看要调用stat（）或fstat（）

### 39.9

删除文件

unlink（“文件名”）

### 39.10

创建目录

mkdir（“名字”，0777）

空目录有两个条目

. 自身

.. 父目录条目

### 39.11

读取目录

用opendir（）

readdir（）

closedir（）

### 39.12

删除目录

rmdir()

在删除该目录之前要求目录是空的（只有“.”，“..”）

### 39.13

硬链接

删除文件用unlink（）

link（）需要两个参数-旧路径和新路径

ln recode.txt rec2

在创建目录中创建另一个名称,具有相同的inode（直到一个文件被修改，修改一个文件inode会改变）

两个文件名之间没有区别

都是指向文件底层元数据的链接

当文件取消链接时

会检查inode号中引用计数reference count

用这个技术跟踪用多少不同的文件连接到这个inode

当一个计数到0时

文件系统才会释放inode和相关数据块

### 39.4

符号链接 symbolic link（软链接soft link）

硬链接有局限

不能创建目录硬链接

不能硬连到其他磁盘分区的文件

ln -s 旧路径 新路径

区别：

链接本身是一个不同类型的文件

使用ls -al时第一个字符为

‘d’是目录

‘-’常规文件

‘l’软链接

修改后原文件也会被修改

删除原始文件，symbolic link的链接会指向不存在路径名

### 39.5

如何从许多底层文件里组建成完整的目录树

先制作文件系统

挂载它们

使其内容可以访问

mkfs-创建文件系统

mount-以现有目录作为目标挂载点（mount point）（将写的文件系统黏贴到目录树的点上）

## 40

### 40.0

vsfs(very simple file system)

unix 文件系统简化版本

纯软件

### 40.1

文件系统的数据结构 data structure

文件系统在磁盘上使用哪些类型的结构来组织数据和元数据

访问方法 access method

如何将解除发出调用-如open（），read（），write（），映射到结构上

### 40.2

将磁盘分成块 block

每个块为4kb

假设有64个块

用户数据（D）

文件系统大多数空间都是用户数据

存放用户数据的区域称为数据区域date region

inode table（I）

要在磁盘留出一些空间来存放

保存磁盘上inode的数组

分配结构 allocation structure

要用某些方式来记录I和D空闲还是以分配

是文件系统必须部分

位图bitmap

有一个空闲列表

指向空闲块

有指向下一个空闲块

第一块-超级块superblock

包含关于特定文件系统的信息

文件系统有多少个inode 和数据块

### 40.3

inode （index node）

这个节点最初放在数组中

在访问特定的inode用到这个数组的index

每个inode都有一个数字（inumber）隐式引用-就是low level name

给定一个inumber就可以直接计算磁盘上相应节点的位置

所有关于文件的消息叫元数据metadata

文件除了存粹的用户数据之外，其他的都叫元数据

间接指针indirect pointer

不是包含用户数据的块

是包含更多指针的块

每个指针指向用户数据

inode有一些固定指针和一个间接指针

如果文件够大，会分配一个间接块去存数据，让间接指针指向他

文件可以增长到1024x4kb

双重间接指针double direct pointer

包含间接间接块指针

这样文件就是1024x1024x4kb

以上为multilevel index多级索引

使用不平衡树是因为大多数数据都很小

大约2kb

### 40.4

目录组织

一个基本目录上包含一个二元组

系统将目录视为特殊文件

目录有一个inode

改目录由inode指向数据块

### 40.5

空闲空间管理free space management

文件系统要记录哪些inode是空的，这样分配目录时可以找到它

如果创建一个文件

要为文件分配一个inode，文件系统通过位图搜索一个空闲内容并分配

将inode标为已用

用最终的消息更新磁盘上的位图

### 40.6

访问路径access path：读取和写入

从磁盘读取文件

open（“/foo/bar”，O\_RDONLY）

文件系统先要找到文件bar的inode去获取基本信息

文件系统必须遍历traverse路径名

所有的traverse都要从根目录的inode开始

根的inode是总所周知的

大多数的根inode为2

递归traverse路径名直到找到所需的路径名

将bar的inode号存入内存为此进程分配一个文件描述符返回给用户

每次读取要先查询inode然后读取

然后更新inode的最后访问的字段

写入磁盘

程序会掉用write

与读取不同，写入文件会重新分配一个块

当写入一个文件时

不仅要将数据写入磁盘

还要先决定哪些块分配给文件来更新磁盘的其他结构

每次写入文件会有5个IO

读取数据位图-更新以标记新分配的块被使用

写入位图-将新状态写入磁盘

读取inode

写inode

写入数据块本身

### 40.7

缓冲缓存

读取和写入是昂贵的

会导致慢速

大多数文件系统积极使用DRAM来缓存重要的块

现代系统采用动态划分dynamic partitioning

操作系统将虚拟页面和文件系统页面集成到统一页面缓存中unified page cache

第一次打开时间会久一点，但第二次打开就会命中缓存，不需要io

但高速缓存不能减少写入流量

写缓冲write buffer

通过延迟写入

文件系统可以将一些数据更新成很多数据（堆积起来）

一次性写入可以节省io

一些程序不喜欢缓冲因为在缓冲时如果系统崩溃数据会丢失

所以会采用fsync强制写入

## 41

### 41.0

老unix文件系统

一小块S-超级块，包含卷的大小，有多少inode，指向空闲列表的头部指针

大一点的inode块-包含文件系统所有的inode

剩余的都是用户数据块

### 41.1

问题

性能不佳

老unix文件系统将磁盘当成随机存取

有昂贵的定位成本

文件数据块离inode块远-需要时间去seek

文件系统最终会变得碎片化fragmented

原始块太小512字节

### 41.2

FFS（fast file system）

让文件系统的结构和分配策略具有磁盘意识

通过保持与文件相通的接口（open， read ，write，close）

### 41.3

结构

柱面组cylinder group（块组 block group）

假设一个磁盘有10个柱面组（有10个group）

每个柱面组都有

超级块super block的副本（每个group都一样）-如果有一个损坏，可以用另一个来挂载

inode位图bitmap-针对每组的inode

数据块 bitmap--针对每组的数据块

位图是文件系统中可用空间的绝佳方法

### 41.4

策略：分配文件和目录（把相关的同学放在一起）

目录放置

找到分配数量少的cylinder group和大量自由的inode

将目录数据和inode放在这个组中

文件

确保将文件数据块分配到与其inode相通的到cylinder group中（防止长时间寻道seek）

与同一目录的所有文件在同一个cylinder group

### 41.6

大文件会妨碍相关文件放在同一个cylinder group中

FFS的做法：

将一定数量的块放在一个块组中

将大文件分开，每一个块组放在不同的cylinder group

### 41.7

当时文件平均大小2kb

块4kb

有内部碎片internal fragmentation

子块sub-block

有512字节

文件系统可以将它分配给文件

如果创建一个1kb小文件，占用两个子块

随着文件变长

会分成更多子块直到到了4kb为止

问题：

效率低下

ffs做法：

修改libc库

将它们缓冲写入

## 42

### 42.0

主要挑战

在崩溃和断电情况下怎么更新持久数据结构

在更新数据结构过程中，突然没电了怎么办？

崩溃一致性问题crash- consistency problem

有两个磁盘

磁盘一次只为一个请求服务

当更新完一个磁盘时，没电了

就会不一致inconsistent

### 42.1

假如有一个文件

其inode里面是

size：1

owner

pointer：4

pointer：null

pointer：null

当更新追加一些数据时

会增加一个块

inode 变成了

size：2

owner

pointer：4

pointer：5

pointer：null

更新的有

数据块

inode

数据位块

崩溃的场景、

只有数据块写入

没有指向，跟没写一样

只有inode写入

会信任一个指针，但指向的是垃圾数据

只有数据位图更新

已经分配了

没有inode指向它

空间泄露space leak

等等。。。

### 42.2

文件系统检查程序

fsck

在文件系统挂载或可用之前运行

超级块

检查超级块是否合理

确保文件系统大小大于分配块数

空闲块

fsck扫描inode，间接块，双重间接块

生成正确版本的分配位图

inode状态

检查每个inode是否损坏

如果存在问题，不易修复，会被清除

inode链接

验证被分配的inode的链接数

重复

fsck检查重复指针

如果指针指向超出有效范围的某个指针

会被认为是坏的

目录检查

确保'.' '..'是前面的目录

确保目录条目的每个inode都被分配

问题：

太慢了

可能要几个小时来扫描

预写日志write ahead logging

在更新结构之前

会写下一些信息-描述将要做的事情

在崩溃发生时

可以查看最新日志

用来重新使用日志里的内容

数据日志data journaling

包含

（逻辑日志logical logging）-提高效率

开头：TxB

结尾：TxE

事务标识符transection identifier TID

（物理日志physical logging）-确切的物理内容

一块inode的最终选址

一块位图的最终选址

一块数据块的最终选址

操作流程

将那些logical logging 和 physical logging 写入日志

将physical logging所对应的信息写入磁盘（加检查点 checkpointing）

如果在写入日志时发生崩溃会很糟

解决：

在写完日志时写上TxE

确保安全

此过程为原子性

最终操作流程

日志写入：写入物理日志

日志提交：加上TxE

加检查点：把原数据写入磁盘

利用日志恢复recover

如果已经提交日志，在检查点发生崩溃（redo logging重做日志）

文件系统会扫描日志

找到已经提交到磁盘的磁盘事物

这些事务会被replayed，重新做一次

使日志有限

如果不断添加事务

会被很快填满

问题：

日志越大

恢复时间越长

无法像磁盘提交事务

文件系统将日志视为循环数据结构

一遍一遍重复使用

日志超级块

标记日志最旧最新的事物

在最后一个步骤加一个步骤

释放：通过更新日志超级块，把一些事物标记为空闲，去减少恢复时间

最终日志协议

日志写入：写入物理日志

日志提交：加上TxE

加检查点：把原数据写入磁盘

释放：通过更新日志超级块，把一些事物标记为空闲，去减少恢复时间

元数据日志metadata logging 有序日志ordered logging

提高性能

消除了用户数据

先将用户数据块写入磁盘

再做以下操作

日志元数据写入：写入inode和位图

日志提交：加上TxE

加检查点：把原数据写入磁盘

释放：通过更新日志超级块，把一些事物标记为空闲，去减少恢复时间

块复用

如果有一个目录foo

它加了一个条目然后删除了foo

这些会被记录在日志里

然后在删除的地方加了一个foobar块

这时发生了故障

所有信息被重放

这时foo的内容会覆盖foobar

解决方案

1.永远不再使用这些块

直到该日志被删除

2.把新纪录添加到 撤销revoke纪录

删除目录会将撤销纪录里的东西写入日志

在重放日志时

会扫描撤销纪录

被撤销的纪录不会被重放

### 42.4

反向指针一致性backpointer-based consistency

每个块都会有一个反向指针

如数据块会有一个额外指针指向inode

可以相互检查

## 43

### 43.0

别的文件系统-日志结构文件系统log- structured file system

原因

内存大小不断增加

宽带传输增加

现有文件系统在许多工作负载上表现不佳

文件系统不支持RAID

写入磁盘时，LFS会将更新缓存在内存段中

当段满了的时候

它会写入磁盘

传到磁盘未使用部分（不会覆盖现有数据）

因为段很大，可以有效利用磁盘

### 43.1

按顺序写入磁盘

就是按顺序写入

inode指向数据块

### 43.2

为了使写入时间变少

用写入缓冲write buffering

在写入磁盘之前

LFS会跟踪内存的更新

当收到足够量的更新时

会写入磁盘

被写入磁盘的那一大块叫段segment

### 43.4

找inode

inode映射 inode map

imap-输入inode号输出inode磁盘地址

每次将inode写入时，会更新imap

映射会在inode写入时一起写入（在旁边）

### 43.5

检查点区域checkpoint regionCR

包含指向最新inode映射的指针

CR放在固定区域

CR每30s会更新一次

### 43.8

储存目录

就是inode映射的集合

### 43.9

处理垃圾

LFS会定期查找文件数据

垃圾收集garbage collection

自动为程序释放未使用的内存

定期读入很多旧段

然后确定哪些块在这个段里面

写出一个新段只包含活着的块

释放旧的

### 43.10

## 44

### 44.0

数据完整性data intehrity

数据保护data protection

### 44.1

磁盘故障模式-要么在工作，要么完全失败

故障停止fail stop模型

潜在扇区错误latest-sector erroes LSE

磁头由于某种原因接触到表面（磁头碰撞head crash）

块错误block corruption

有缺陷的磁盘会将块写在错误的位置

出现故障时

磁盘无法报告问题

### 44.2

处理潜在的扇区错误

利用冗余

在RAID 4-5级中

会读取其他磁盘

计算缺失值

重建reconstruct磁盘

### 44.3

检测corruption（如何阻止用户读取错误数据，如何知道块是坏的）

校验和checksum

有一个函数

该函数以一块数据（4kb块）为输入

计算这段数据

产生小数据内容（4b）

与数据一起储存

常见的校验函数

用XOR计算（简单）

每个块进行XOR运算

将块里的数据进行排列

每一列进行XOR

问题：

一列两个错误就会检测不出

加法（简单）

快

进行二进制补码加法

如果数据移位，不好

Fletcher检验（复杂）

用两个校验字节计算

s1，s2

s1=s1+ 一个字节%255

s2=s2+s1 %255

循环冗余校验CRC

将数据块视为大的二进制

将其除以约定值k

除好的值为CRC值

如何在系统布局校验和

基本

为每个数据块旁边添加一个校验和

别的方法

将校验和打包到512字节的块中

有一坨校验和放在一起

后跟一坨数据块

效率低

在更新数据块时

要先跑到校验块那先更新

在更新数据块

### 44.4

使用校验和

客户端在读取数据块时，也会从Cs（D）里读取校验和-储存的校验和（stored checksum）

客户端计算读取的块D上的校验和-计算的校验和（computed checksum）

之后比较stored checksum和computed checksum看看是否相等

### 44.5

现代磁盘有几种的故障模式，要有不同的解决方法

错误位置写入misdirected write

它们正确的将数据写入磁盘

但位置错误

处理错误写入

为每个校验和中添加信息

加物理标识符physical identifier ID

如disk几，block几

如果信息不匹配，错误写入发生

冗余时错误检测和恢复的关键

### 44.6

丢失的写入lost write

当设备通知上层已经完成

但这个数据其实是未持久

处理丢失的写入

写入验证write verfy/写入后读取read after write

通过写入后立刻读取

确保数据在磁盘表面

此方法慢

方法二

在文件系统每个inode和间接块之间

包含检验和

如果数据丢失

inode内的校验和就会无法匹配

### 44.7

大多数数据很少被访问

磁盘擦净disk scrubbing

通过定期检查所有的块

检查校验和是否有效

典型的系统会每晚或每周安排检查

# Distribution

## 47

### 47.0

分布式系统

当web连接其他地方的web服务器时-简单的client server 分布式系统

google 不只是一台机器进行交互

是用大量机器来提供的

每台机器提供站点的特定服务

故障failure

机器，磁盘，软件都会故障

通过大量机器来开起来很少失败

性能performance

人员必须仔细考虑如何完成任务

减少发送信息的数量

安全security

当连接到远程方时

确保远程方是他们声称的那些人

### 47.1

通信communication

通信基本是不可靠的（丢失，损坏，无法到达）

在电气或其他问题时-位会被反转

网络交换机，路由器，终端缺少缓冲-数据包丢失

如果许多数据包同时到达

路由器内存是无法容纳所有数据包的

会选择丢弃drop

### 47.2

几乎所有的现代系统中，都有UDP/IP网络栈

UDP

使用套接字（socket）API-创建通信端点communication endpoint

其他机器的进程将UDP数据报datagram发送到前面的进程

使用它会数据包丢失

但包含checksum，检验包是否损坏

### 47.3

可靠的通信层

发送方如何知道接收放实际收到了消息

确认acknowledge

发送方向接收方发信息

接收方发短消息确认收到

超时timeout

当发送方发送消息时

会在一定时间关闭

如果在此期间没有收到消息

会重试retry

（发送方要保留消息副本）

问题：

如果接收方会消息了但发送方没收到

接收方可能会返回两次

只接收一次exactly one

发送方为每个消息设置一个ID

成本高

顺序计数器sequence counter

发送方和接收方每一方会维护一个计算器

使其值为一致

设计数器值为1

发送消息后变成2

接收方（1）会受到这个值（1）

相等接收方变成2

如果又收到计数为1的会拒绝

### 47.4

通信抽象

分布式共享内存Distributed shared memory （DSM）

使不同机器上的进程可以共享一个大的虚拟地址空间

在计算机访问时有两种情况

1.页面已经是机器上的本地页面

2.页面在其他机器上

最大问题-处理故障

所有机器为一个地址空间

如果有一部分数据结构不可用了

如果下一个指针指向消失的空间

那很难找到是哪个机器的

性能不佳

几乎没人用

### 47.5

远程过程调用remote procedure call（RPC）

使在远程机器上执行代码像调用本地函数一样简单直接

进行一个过程调用，在一段时间后返回结果

两部分：

1.存根生成器stub generator

通过自动化

消除参数和结果的打包

当用代码调用函数时：

创建缓冲区-某种大小的连续字节数组

将所有消息打包到消息缓冲区中-包含要调用的函数的标识符，参数

将消息发送到RPC服务器

等待回复

解包 返回代码 参数

返回调用者

（服务器）

解包消息

调用实际函数

打包结果-返回参数，放入回复缓冲区

发送回复

2.运行时库run-time library

处理性能和可靠性问题

如何找到远程服务

客户端必须知道运行所需RPC服务器的主机名和IP地址和端口号

用域名

一旦客户端知道与哪个服务器通信

获得特定远程服务

构建RPC的传输协议

是用TCP/IP-可靠协议

会导致性能低效率

要发送两个额外消息（确认相应，确认接收）

还是用UDP/IP-不可靠通信层

更高效

## 48

### 48.0

分布式文件系统

有许多客户端机器和文件系统

服务器将数据存在它的磁盘上

客户端用协议去请求数据

允许客户端共享数据

集中管理centralized administration-备份文件不需要众多客户端，只要一点服务机器

安全security

### 48.1

基本

比文件系统有更多组件

客户端

用客户端文件系统client-side file system访问目录

系统调用（open，read，write，close，mkdir）

系统提供对文件透明transparent访问

用户发出read（）

客户端文件系统向服务端文件系统发送消息

服务器将从磁盘取出块

发送消息，将请求数据返回客户端

客户端将数据复制到用户的缓冲区中

之后如果客户端重复访问，就会命中，不会产生流量

### 48.3

NFSv2协议（简单快速的服务器回复）

服务器关闭一分钟都会让客户感到不快

### 48.4

无状态协议stateless

服务器不会追踪每个客户端发生的事情（不知道客户端在缓存哪些块）

每个客户端操作必须包含完成请求

有状态协议stateful

给定路径名

服务器返回文件描述符

以后进行相应的读写操作

问题：

如果在返回文件描述符以后

服务器崩溃了

在重启服务器之后

文件描述符就没有了

恢复协议recovery protocol

客户端必须确保在内存保存足够信息

以便告诉服务器它知道的位置

问题2:

当服务器打开文件时

客户端必须用close来关闭文件

如果客户端崩溃了

服务器永远也收不到close

### 48.5

NFSv2

文件句柄file handle

用于唯一的描述文件和目录

3个组件

卷标识符-通知服务器请求指向哪个文件系统

世代号-复用inode号时递增它

inode号-告诉服务器请求访问该分区的那个文件

协议

lookup获取文件句柄（/）

如果客户端运行应用程序打开/foo。txt

会向服务器发出查找请求

向服务器传递文件句柄和名称

如果成功，返回foo。txt的句柄（/foo。txt）

客户端向系统发出read write

### 48.6

如果要访问/foo/bar/foo。txt

需要让客户端发送3次lookup

一次在/找foo

一次在/foo找bar

一次在/foo/bar找txt文件

### 48.7

当客户端向服务器发送消息时有时候不会收到回复

处理故障（客户端）：

重试请求

在发送请求之后

客户端将计时器设置为在指定时间之后关闭

如果在关闭之前收到回复

取消定时器

幂等的idempotent

操作执行多次和效果与执行一次的效果相同

如果储存3次和储存一次效果一样

那将值存到内存中时幂等的

lookup和read是幂等的

重复写入没有关系

### 48.8

改善分布式文件系统性能

客户端缓存caching

第一次访问是昂贵的

之后几次就会块（在缓存里）

可以用作写入零时缓冲区

将应用程序的write延迟与实际写入分离

程序对write调用会立即成功

### 48.9

缓存一致性问题

假设有两个客户端c1，c2

c1向服务端发送write

但数据会先放在缓存里而不是直接写入

这时c2向服务器发送read

读到的还是旧数据

更新可见性：

来自一个客户端的更新，如果被其他客户端发现

陈旧的缓存stale cache：

在c1覆盖数据前

c3先读取了数据

那在c3里缓存的数据是旧数据

解决更新可见性-关闭时刷新flush-on-close：

当应用程序写入文件并关闭时

客户端将所有更新刷新到服务器

解决陈旧的缓存stale cache：

客户端会先检查文件是否已更改

再使用缓存的内容

在打开文件时

客户端系统会发出getattr获取文件属性

属性包含服务器上次修改文件的信息

如果修改时间比读取时间晚

客户端会让文件无效invalidate

将它从缓存中移除

但让客户端一直发送getattr也不好

属性缓存attribute cache

首次访问文件时

文件的属性会放在缓存中

在一定时间后超时

### 48.11

假设就数据为

a

b

c

新数据是

x

y

z

客户端向服务器发送一个write请求

服务器接收此消息并发送到磁盘

向客户端发送成功

客户端发送第二个write请求

这次服务器把信息放在缓冲中并返回成功

但在写之前服务器崩溃了

重启后服务器接收第三个write请求并返回成功

之后内容如下

x

b

z

解决：

每次会提交到稳定（持久）储存

## 49

### 49.0

Andrew 文件系统

扩展scale

### 49.1

AFS v1

原来叫ITC分布式文件系统

后来重新设计和最终协议

基本原则

在访问客户端的本地磁盘local disk时，进行文件缓存whole-file cacheing

当open（）文件时

将从服务器读取整个文件

并存储在本地磁盘文件中

后续应用程序的read write操作会定向到本地文件系统（不用网络通信）

close文件时

文件会写回服务器

（NFS是缓存块而不是整个文件，并缓存在客户端内存中而不是磁盘）

当客户端调用open时

AFS客户端向服务器发送fetch协议请求

fetch协议将整个文件路径发送给服务器

服务器将整个文件发送给客户端

客户端将文件缓存在磁盘上

AFS用客户端内存缓存块副本

完成后检查文件是否已被修改

如果被修改用store协议将新协议写回服务器

### 49.2

AFS v1两个主要问题

路径查找成本过高

客户端将整个路径名发给服务器

由于客户端同时访问服务器服务器花费大量cpu时间

客户端发出太多testauto协议消息（跟getattr相似）

用该协议消息生成大量流量

服务器花费大量时间告诉客户端时候可使用文件的缓存副本

解决：

设计可扩展协议

### 49.4

AFS v2

回调callback

当客户端缓存的文件被修改时

服务器将通知客户端

（客户端不需要在联系服务器）

文件标识符file identifier。FID（类似于NFS的文件句柄）

不是让整个路径名发送给服务器，让服务器找

而是让客户端沿着路径查找，每次一个，缓存结果

列：

客户端访问/foo/bar/foo。txt

客户端先获取foo内容

将他们放在本地磁盘缓存中

参考书445页

### 49.5

缓存一致性

不同机器上的进程一致性

当新文件刷新到服务器时

服务器会中断拥有缓存副本的客户端的回调

确保不再读取过时副本

之后客户端需要让服务器重新获取文件新版本

有时不同机器会在同一时间修改文件

用最后写入者胜出last writer win

最后调用close就能更新文件

### 49.6

崩溃恢复

假设client1在本地磁盘上缓存了F

client2更新了f

此时c1正在重新启动

服务器无法给c1发送消息

c1会丢失这些数据

解决：

在重启时

c1将所有缓存内容设为可疑

在访问时会向服务器询问

### 49.7

AFS与NFS比较

大文件顺序重新读取

AFS比NFS块

AFS用本地磁盘缓存

NFS只缓存块

文件覆盖

AFS客户端需要完整提取旧文件再覆盖

NFS只覆盖块

NFS块

在大型文件里访问一小部分块

NFS快

AFS要获取整个文件

NFS基于块协议，执行IO与读取或写入大小成比例