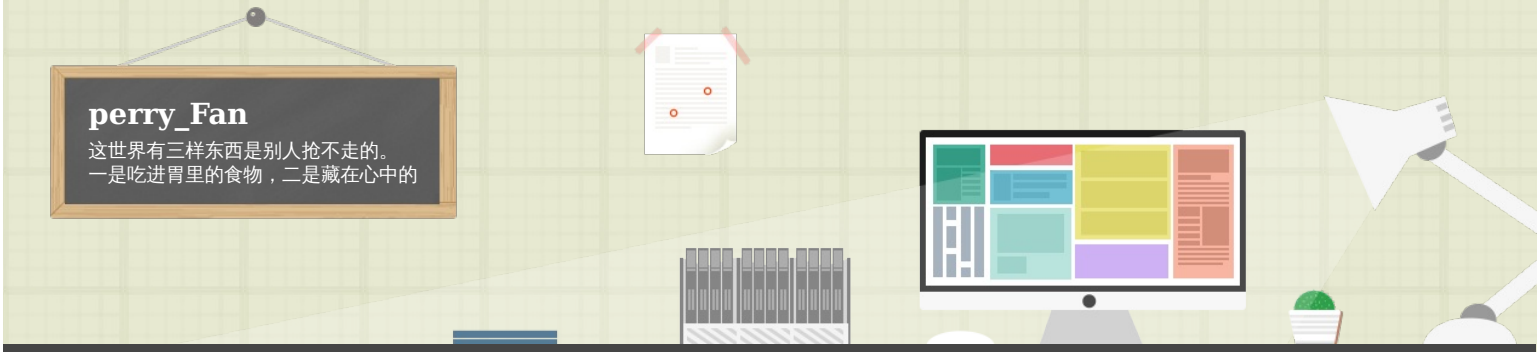


原 30分钟让你了解操作系统基本概念

2016-05-17 15:04 2897人阅读 评论(0) 收藏 举报

分类： 操作系统（1）

前言：本文是《操作系统教程（陈怀临注释）》的读书笔记，陈首席是在原书pdf 图片上注解，字体比较模糊，故我把注释中觉得比较重要的片段摘录下来。读完此文可以让非技术人员对操作系统有框架性的认识，也可以唤起技术人员记忆中某些概念片段，实际上很多概念我也理解得有点模糊，大家一起学习。

1、OS其实是概念多于理论，技术多于算法。因此把握OS最重要的是把握概念，特别是概念的层次化。

2、OS的本质是实现了CPU, Memory的虚拟化。现代的虚拟化技术是再次虚拟化，实现了OS的虚拟化。

3、在经典Unix下，kernel是共享的，因此必须互斥。kernel属于每个（each）32bit进程地址空间的一部分，例如3~4G部分。

4、Page on demand 是理解VM虚拟内存的Rule 1（Swap）。

5、在经典 **OS**设计中，例如**Unix**， **“Everything is a file”** 是非常重要的设计原则。任何一个外设，最后都通过文件系统来表达。一个通过**open**得到的文件句柄可以唯一的定位一个设备（块设备or字符设备），并可以通过文件的 **read／write**来操作。

6、初学操作系统的大学生通常会对文件句柄（File Handler）有点迷惑。可以这样理解。就是open（）的时候，操作系统为你构建一个表项的数组的下标。这样也就理解了一个进程可以打开的文件数目是有上限的。为什么？数组的大小是固定的，除非改参数。

7、**OS**最重要的概念就是进程（ **Process**）。可以理解为是操作系统**“**管理 **”**的最小单位。虚存（**VM**），文件（ **File**）都属于（**Belong to**）这个进程的 **domain**。例如，属于哪个进程的。进程就是一个在运行中的程序，通常是一个 **ELF**的加载。

8、学习文件系统的时候，不要去纠结驱动程序的实现。类似谈恋爱，非要大家都坦白从幼儿园开始的情史，是大家彼此过不去。要学会**“**透明**”**。概念到文件系统，就刹住。否则，为了理解文件系统，非要把 **INT13**通读，是没有必要的。文件就是文件。

9、文件系统最重要的是控制块（ **Control Block**）。要知道数据（例如， **512B**）在硬盘哪个地方。而且要靠指针串起来。例如，早期**DOS**的 **FAT** 表都是这个目的。在现在分布式文件系统中，称为 **metadata**。目的都一样：在哪里。 **metadata**或者**control block**失效了，数据都无法定位了。

10、基础 教材通常会有意识 的凸显概念。其实任何概念本身就是抽象和总结 出来的。什么是 **“** 虚拟处理器**”**。说白了，就是每个进程数据结构里的 **CPU** 相关寄存器的值。那就是对于那个进程而言，她的虚拟处理器。虚拟内存？就是她的，例如，页表和 **MMU**的设置。

11、初学 **OS**的同学不要去过分理解虚拟处理器这个概念。还是应该从经典分时系统出发。现代 **OS**的本质是分时。其他都是演变出来的。分时就是大家皇帝轮流做。因此下台的时候要保存一些状态。等下次轮到时，从上次断的地方重新来。

12、输入输出（ **I/O**）的访问必须串行化（**Serialization**），否则就乱了套。 **why**？写过驱动就知道，控制设备的那些 **control register**（控制寄存器）还没有完成一个操作，如果被覆盖，设备就死机或者 **reset**了。并发是 **CS**许多算法的目标，但底线是：和串行语义要一致。

13、操作系统另外一个重要任务是参与和指导 **CPU**设计。现代**silicon design**从来都是**co－design**。否则，硬件工程师都不知道在干嘛。不能画电路图玩吧。真正懂一个 **silicon**的必须包括**OS architect**。这也是为什么 **OS**是计算机科学 **or／and**工程的美丽之花。

14、在单 **CPU**的年代，除了中断（时钟，外设），一个计算环境不存在并发。 **OS**做调度也是在几个固定的点，例如， **timer**，**syscall** ，**wait for I／O** 等。

15、**MultiTasking** 的本质就是大家共享资源例如， **CPU**。例如，通过分时，或者是主动退让（ **Yield**）。多道（ **Multi Tasking**）和多重（**Multi Processing**）处理的区别是： **multitasking**就是一个**CPU**，例如。 **multiprocessing**是多个**CPU**。现在的多核，多（硬件）线程都属于这个范畴。 **MultiTasking／SingleCPU**本质上还是串行化的（**Serialized**）。

16、在学习操作系统的时候，一个重要的概念是传统操作系统内核是独占，不可剥夺的， **Kernel is not preemptive**。这个概念的理解把握对阅读源码，理解 **Unix／Linux** 的演化是至关重要的。对锁机制，锁粒度的优化也是最重要的。

17、用户态／核心态的本质是： 保护。保护什么？ **Kernel**的全局变量。为什么？ **Kernel**是共享的。每个进程，例如， **32**位系统**Linux**，是**4G**空间。**3G**用户＋ **1G**核心 ＝进程。因为是共享的 **kernel**，所以需要互斥。否则，全局变量用一半就被冲了。

个人资料

范培华

关注 发私信

访问： 38030次
积分： 750
等级： BLOG 3
排名： 千里之外

原创： 54篇 转载： 12篇
译文： 3篇 评论： 18条

文章搜索

文章分类

Android应用开发 (22)
Android高级开发之路 (1)
Python初窥门径 (0)
Android问题 (8)
Android组件 (3)
算法与数据结构 (2)
Java (1)
职业发展 (4)
前端开发 (8)
设计模式 (1)
数据库 (1)
操作系统 (2)
网络基础 (1)

文章存档

2016年12月 (1)
2016年11月 (1)
2016年10月 (2)
2016年09月 (1)
2016年08月 (4)

展开

阅读排行

给 Android 开发者的 RxJav... (3377)
如果有人问你数据库的原理，... (3102)
30分钟让你了解操作系统基... (2893)
缓存淘汰算法--LRU算法 (2693)
知乎日报 API 分析 (2489)
Android 6.0中"Unable to f... (2109)
HybridAPP框架MUI（跨平... (1329)

18、理解 **kernel**空间是**PART OF** 一个进程空间，是对现代操作系统把握最重要的概念之一。例如，经典 **OS**有一个重要的**statement**：**kernel**是没有**context**的。什么意思？ **kernel**不存在生命。是属于一个进程的，而且是共享的。

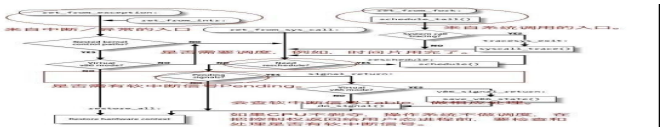
19、系统调用是操作系统里略微难理解的一个概念。其实就是通过一个特殊指令，使得 **CPU** 陷入到异常处理，然后通过查表（事先填好），最后调用相应的 **kernel**库函数。（在经典**os**里）， **kernel**就是全局变量＋函数。写系统调用时，要注意的是对参数传递的约定要比较清楚。

20、操作系统的发展经历了单一内核（ **Monolithic Kernel**）和微内核（**Micro Kernel**）的学术争论。最经典的是 **Linus**和操作系统泰斗**Andres T 1992**年的辩论 **(http://t.cn/S4dT62 -Torvalds_debate)**。 现在基本上是 **convergence**，融合了。特别是在虚拟化技术的今天。

21、进程是最小管理单位；（ **System Scope**）线程是最小调度单位。同一个进程的线程序共享内存，例如全局变量。通常说的线程在**kernel**里对应一个调度**object**，通常称呼这样的 **thread**叫做**System Scope**。如果是 **Appliacation Scope**，叫做用户线程，在**kernel**里不存在 **entry**
。
22、在理解操作系统的时候，内核（ **Kernel**）是属于一个进程（**Process**）空间的一部分 是一个重要的概念。你编的代码＋内核 构成了一个进程空间。
23、操作系统从过去的 **Monolithic OS**与**Micro Kernel** 的学术和工程对立，发展到互相融合。特别是最近的虚拟化技术的发展，使得已经无人再去纠结这些结构之争。 **CPU**，多核技术的发展也是导致这些区别淡化的重要原因之一。

24、学习 OS的时候，会接触CPU，例如 x86. 其实x86 不太适合做 CPU教材例子。 IA32为了支持许多历史设计含有了许多老东西，例如段地址等。现代 CPU，包括Intel ，都已经是采用 Flat Memory结构。学生们没有必要去知道和理解之前的技术。应该直接一步到位。
25、中断／异常处理是掌握一个 CPU重要的部分。不同的CPU实现都有细微的区别，但本质都类似。 保护现场，处理中断／异常，根据情况决定是恢复现场执行，还是 reboot CPU。

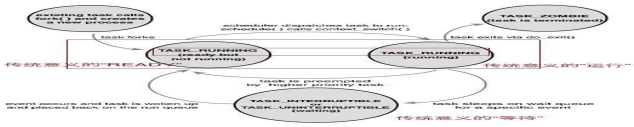
26、信号在 Unix中比较绕，是异步的通知，进程之间通过互相发 kill()。也可以是 kernel发给一个进程。有的signal不能被捕获，进程必须消亡；有的可以自己接管，接着运行。看是否有 signal的时机需要很清楚：异步方式：在系统调用，中断返回的时候。
27、如图所示， 操作系统从中断，异常，或者系统调用进来，要结束的时候，都存在一个检查点（ checkpoint），看看是否需要调度，是否需要处理软中断（ signal）。



28、中断处理程序要求的是快速。因此 Linux里有一套完整的Bottom Half的机制来做优化，把不存在重入危险的，不是那么急需立刻需要处理的，可以通过 Bottom Half的一些手段，来“延迟 ”执行。从而使得整体系统并发度提高。早期 BH，TaskQueue 机制都已经不用了。
29、在学习 2.2中断技术方面，可以不要太陷入 Linux独有的中断技术细节中。大概了解一些概念就可以了。把握两个重要的概念：中断上下文；进程上下文。中断处理程序需要快，然后迅速开中断，防止系统堵塞。这是Linux各种各样的 Bottom Half机制的由来。

30、传统 Unix的内核是非剥夺式的调度（ non preemptive），目的是简单从而保证内核数据结构一致性（内核是各个进程共享的地址空间）。现代 OS，如 2.6之后的Linux ，可以支持内核 preemptive schedule了。目标是支持高并发系统。
31、fork, pthread_create 最后都是调用clone()函数，传递进去的参数不同而已，子进程/线程在内核中都是产生一个task_struct 元素。"CLONE_THREAD"参数可以告诉clone调用，这个新产生的object是个thread逻辑。从而在thread_group 里会把这个新产生的object加入，作为一个线程来对待，但这种对待是一种逻辑理解而已。

32、例如，一个时间很长的读磁盘的操作。在阻塞sleep的时候如果是可中断的，另外一个进程可以发signal，这样这个进程就能够从这个读数据的系统调用中退出来了。



33、为什么需要线程（ thread）来支持并发？都用进程不就可以了嘛？因为属于同一个进程的多个线程是共享全局变量的，共享内存的，可以合作来完成一个任务。而如果拆分为多个进程，通常就需要 IPC通信才能完成这样的任务。这就是线程的好处。
34、
a、调度算法学习中要注意“死锁 ”（deadlock ）与”饿死 “（Starving ）的区别。饿死，就是低优先级的任务一直无法得到cpu，一直有高优先级的任务占用cpu。
b、Hard Realtime不一定调度快，是指调度系统保证不错过 Deadline；Soft RT不一定慢，是指调度系统保证错过 Deadline。
c、有兴趣的同学看看NASA的Mars Lander 的优先级翻转的事故。（这往往出现在一个高优先级任务等待访问一个被低优先级任务正在使用的临界资源，从而阻塞了高优先级任务；同时，该低优先级任务被一个次高优先级的任务所抢先，从而无法及时地释放该临界资源。这种情况下，该次高优先级任务获得执行权。）

35、Linux 2.6的调度是一个分水岭：支持 Kernel内部可剥夺。支持SMP下O(1)算法。忘却具体技术细节，把握： 1. 只要数据结构不存在非一致性，就可以开中断和支持被剥夺调度，否则通过锁来保护。 2. 多个Queue一定好于一个 Queue。每个处理器有一个任务队列。
36、现代multiCpu系统许多采用 CPU Affinity的调度方式，就是把task固定分配给某个或某些cpu上。
37、一个应用系统的设计应该是event driven。没有事情的时候，任何线程不能去空转，应该block等待在一个事件上。
38、同步是为了协同工作。通信分为：共享内存（**memory**）方式 和基于消息（ **message**）方式。通信可以来实现同步。例如， **send／recv** 。死锁和饿死都是调度要避免的。

39、学习同步和并发控制时，要注意几个要点： **1.** 同步原语都是等价的。 **2.** 为什么需要原子性？因为，一个简单的 **value＋＋**在指令级别时需要**3**条指令才能完成：从内存中 **Load**；寄存器操作加一。 **Save** 到内存中。在 **3**个指令之间，中断可以任意发生。形成并发。

40. 同步机制可以通过 **Sem**（信号量）。但语义较低。需要在共享内存模式下（例如，多线程），写比较难的控制代码。容易出错。管程（**Monitor**）是一等价的同步机制。把共享数据和访问的代码封装为对象。提供一个一致的同步 **API**接口。语义友善。**Java**等编程语言对 **Monitor**机制的支持已经很完善并被广泛使用 **,**例如**"Synchronized"**的关键字的使用。

41、**Monitor**是一个很有用的机制，特别是在面向对象的程序语言中，都提供了相应的机制。其特点是：通过 **sem**，**mutex**的封装，使得程序员不需要过多的关心低级同步原语（ **raw primitives**）的使用，从而可以在比较高的语义上把握同步，并专注其应用问题本身。

42、**System V IPC** 通信机制是实现多进程之间通信和同步的重要手段。例如，**message queue**， **shared memory**，**pipe**等。其重要前提是：不同地址空间上的多进程之间的通信和数据交换。 **System V**的**IPC**用的比较少了。重要原因是：多线程的同一进程编程模型的流行。

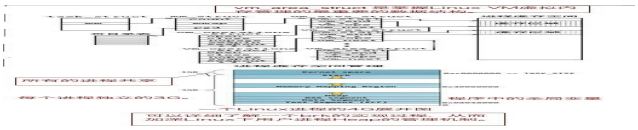
43、**DeadLock** ， **LiveLock**， **Starvation**都是在并发编程中需要注意的事情。 **DeadLock**需要避免进程之间互相要锁然后都进入了等待睡眠的状态；**LiveLock**要避免虽然各个进程不都在睡眠状态，但都在原地抖动 **,**停止不前；例如，吃饭时互相谦让谁先动筷子 **,** 结果谁也没有吃**,**在无解的谦让；**Starvation** 饿死的场景很直接，避免资源的不公平利用 **,**例如**,** 不能某些任务总是获得资源 **,** 有些任务即使长期在等待下 **,** 却没有被分配到资源。

44、

a、**Sem**和**Spinlock**的用法区别在于：如果等待资源时间短和可预期，可以用自旋锁；否则用 **Sem**，通过睡眠／唤醒来处理。 **Sem**由于涉及队列操作，系统存在不确定的延时效率问题。

b、在多核多**CPU**情况下，关中断只能排除当前 **CPU**的并发，只有通过另外一个全局自旋锁的介入，才能封住其他 **CPU**的竞争。

45、编译和链接之后的可执行程序ELF中的地址都是虚拟地址，或者说逻辑地址。在运行的时候通过加载，地址转换，来确定具体的物理地址。cpu指令的执行基于虚拟地址（逻辑地址），是可以通过不连续的物理页面，"营造"成一个连续的逻辑空间的重要保障机制。逻辑地址必须是连续的，物理地址可以是page和page散开的。



46、掌握现代操作系统内存管理系统时，可以把握几个基本知识点。

a、了解**ELF**展开后的格式。 **TEXT**，**DATA** ／**BSS**／ **Stack**／**Heap** 的关系。

b、任何一个 **CPU**的**load** ／**store**操作都是基于逻辑（或者说虚拟地址），通过**MMU**转换一次，成为物理地址，完成**"**动态**"**定位。

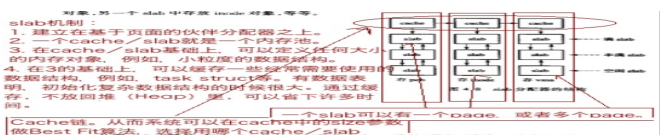
47、

a、连续地址空间管理，通常也可以理解为 **"**堆管理**"** – **Heap**。也可以逻辑地址的堆管理，例如进程的 **Heap**。也可以是一块连续的物理地址空间，例如， **Linux Kernel**为**Slab**分配器提供连续物理页面的内存。

b、内存管理要注意两种碎片： **External Fragment**（一段时间过后无法分配连续的大片内存） 和**Internal Fragment**（分配的内存比实际需要的大很多）

48、

a、基于**4K**大小的页面（ **Page**）的分配粒度太大，**Linux Kernel**的**Slab**机制就是为了实现细粒度内存频繁分配和释放的一种 **memory pool**的机制。 **(2).** 通过架在基于 **Page**的**Buddy**算法内存管理之上， **Slab**可以不需要频繁的把常用的数据结构来来回回放回 **HEAP**里，从而提高了效率。



49、进程切换的时候，要切换页表基址寄存器，不同的进程有不同的页表。基于页表的虚--实转换是一个"算法"，是一种mapping，把相应的bits拿来当索引index。不同进程的虚拟地址Va, Vb可以在各自的page table 里指向同一个物理地址，比如父子进程共用代码段。

50、

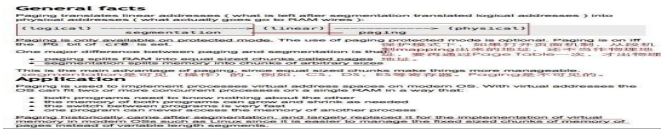
a、每个进程都可以有相同的虚拟地址，例如 **Va**。因此每个进程都必须有自己的页表，从而无歧义的完成虚实转换。

b、多级页表可以使得页表空间不需要连续物理块，例如，二级页表，可以 **Page by page**的分配。

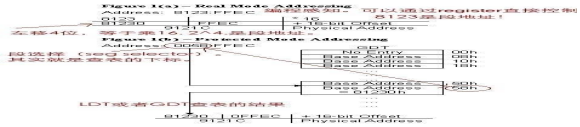
c、通过进程**ID**，可以避免全局页表项目的二义性。

51、分段管理与分页管理是区别和联系是：段是应用编程可感知的；页是应用不感知的，段是早期，例如 **intel 80286**之前的内存管理；**80386**之后有了分页（ **Page**）了。可以**“**基于分段的分页内存管理 **”**。使得在段的基础上再加入页面机制，使得内存使用粒度更加小。没有分页之前， **Intel LDT**的目的类似之后的**Page Table**。每个进程有自己
的**LDT**从而管理自己的物理内存。全局的上通过 **GDT**，各个进程之间共享。有了分页后，段基本上不用了，通过把 **seg selector**都设为**0**，大家都在一个段上，从而，例如， **Linux**，都（只）用分页机制了。

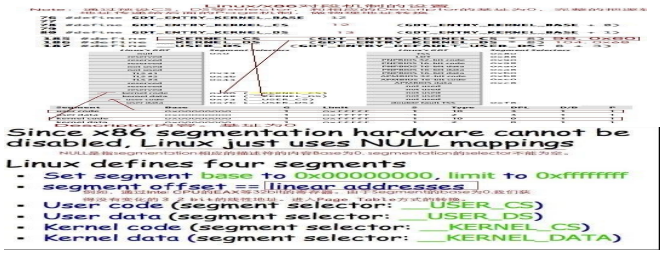
附图是一个比较好段和页机制的关系图。在段做一次映射之后（通过寄存器索引做一次 **LDT**和**GDT** 查表），如果页机制打开的，查表出来的地址不被当作（ **not treated as**）物理地址而是再做一次基于 **Page Table**的查表。然后出来的数据才与 **offset**合并为物理地址去内存取数据。



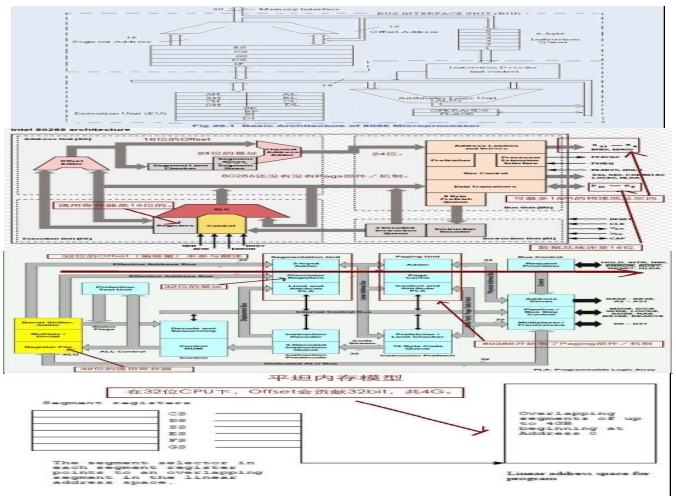
52、现代 **CPU**，例如，**Power**， **PowerPC**，**MIPS** ，**ARM**，都没有段（**segmentation**）机制，而是直接采取不需要编程感知的 **Paging**虚拟内存机制了。段可以认为是 **x86/IA32**的一个过渡历史技术了。



53、Intel CPU 的段模式不能关闭。那么如何无缝的略过，只用页机制呢？ 是通过把 **CS**，**DS**等段选择符预先设置好（参阅宏定义）。然后把**GDT**里面的描述符都手工设置基址为 **0**，大小是**4G**。由于基址为 **0**，通过段机制出来的逻辑地址就没有受到任何影响，完整的进入 **Page**机制转换。

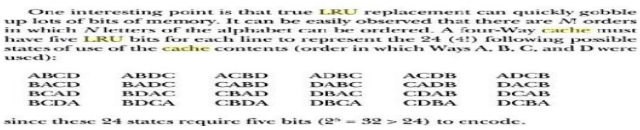


54、技术的发展都是演变的。 **8086**和**80286** 是**16**位机 **GPR**）。地址总线是 **20**位和**24**位。要通过段 **reg**＋**Offset**来**“**拼凑 **”**一个**20**／**24**的物理地址。 **386**是**32** 位机器了。清一色 **32**位；有了**Paging**部件。段部件的偏移量也是 **32**位。所以只要段基址为**0**，就可以使用基于 **Paging**的**Flat** 内存模型。



55、**Paging on Demand**是现代通用操作系统 **VM**管理重要的机制。是一种滞后算法，当需要时来完成虚拟页面（ **Page**）和物理页框（**Frame**）的分配，提高效率。但数据通信系统中，为了避免 **Page Fault**带来的延迟和不确定性，往往是事先完成页框的分配从而确保报文处理的实时性。

56、页面替换算法只有知道未来页面使用情况，才能达到最优。所以，**OPT**（**OPTimal Replacement** ）是一个理想。现实中，有 **LRU**和相应的近似算法（ **NRU**，**NFU** ），**FIFO**， **SCR**，**Clock** 和各种变种。其中 **LRU**和**Clock**比较好。在**CPU**的**cache**设计中，芯片的 **cache set**的算法说 **Pseudo－LRU**。**LRU**代价比较大。需要维持所有页面使用情况来定位 **“**最近没有被使用的页面**”**。例如如果有 **4**个页面其使用情况时**4**！＝ **24**种。如果是**8**个页面，使用顺序有 **40320**种。想想**256**个页面，代价太大。现实系统中，需要实时的，多采用一些近似算法，例如“尽可能最近没有使用的”。

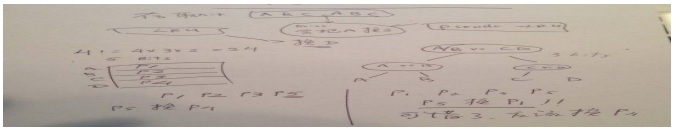


One interesting point is that true LRU replacement can quickly gobble up lots of bits of memory. It can be easily observed that there are $N!$ orders in which N letters of the alphabet can be ordered. A four-Way cache must have five LRU bits for each line to represent the 24 (4!) following possible states of use of the cache contents (order in which Ways A, B, C, and D were used):

ABCD	ABDC	ACBD	ADBC	ACDB	ADCB
BACD	BADC	CABD	DABC	CADB	DACB
BCAD	BDAC	CBAD	DBAC	CDAB	DCAB
BCDA	BDCA	CBDA	DBCA	CDBA	DCBA

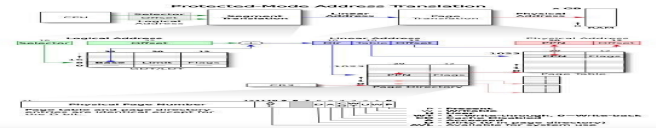
since these 24 states require five bits ($2^5 = 32 > 24$) to encode.

57、在**CPU**的 **Cache**替换算法中，常采用**Pseudo – LRU**。就是一个近似算法。**“**差不多就行了**”**。换出去的一定不会是最近刚用的，但确实可能不是最不用的。如图， 在 **ABC**都是**hit**引用之后，如果有 **miss**，精确**LRU**算法应该是把 **D**换掉［**D**就没有用过］。但如果是 **PLRU**算法，会换了**A**。



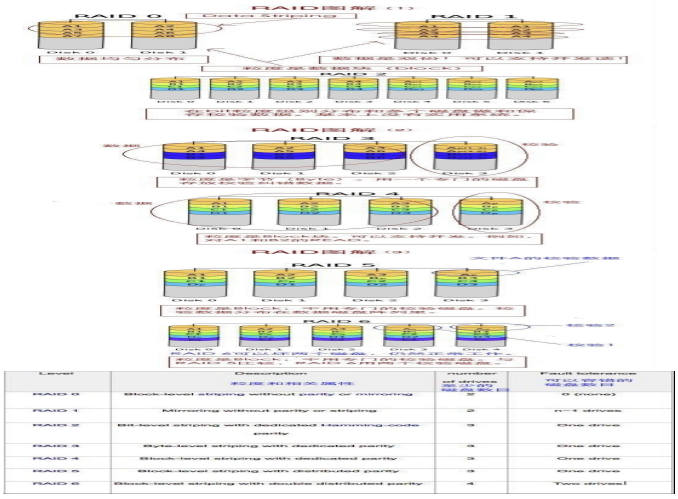
58、局部页面替换算法概念很直接。就是每个进程的页面替换不影响其他进程的。这样可以防止系统的颠簸。 **Working set**的变种叫做**Aging**算法。目的都是一个，逐步收敛和定位一个被替换的页面，例如某个算法中衡量参数的最小值。 **Aging**中每次右移一个**bit**，就是一种衰减过程。

59、内存管理与 **CPU**联系比较紧密。不同的**CPU**都略微不同。对通用系统而言， **Paging on demand**；**copy on write**是操作系统算法层面比较常用的。但对专用系统并不合适；另外，由于启动虚拟到物理的映射，使得操作系统存在一个机会干预内存的分配，保护，从而可以指导策略。



60、**“Everything is a file”** 是现代操作系统的一个重要概念。通过文件系统的抽象和接口来屏蔽设备多线性。理解设备时，要站在总线角度看：**CPU**和其他设备，例如 **ASIC**芯片，都是设备。都具备计算处理能力。需要和内存打交道，大家共享总线。
61、设备通常是 **Memory Based I/O**。控制**Reg**和缓冲区被映射到**CPU**地址空间，例如通过**PCI**。读写控制寄存器和缓冲区通常与 **DRAM**内存一样。通常设备地址 **map**到高端地址区域。通信系统里，还要事先设置一些**cache**属性等，例如，**noncacheable**。总之，通过系统总线控制器或者**MMU**存储子系统部件来调控。

62、RAID磁盘阵列是通过多个小的廉价的磁盘的组合，可以比一个单一的昂贵的大磁盘提供更好的性价比。例如并发的操作和容错。 RAID 0是数据分布； RAID 1是备份镜像；之后的都是有校验纠错磁盘的。 RAID 6是基于数据块和分布式存放纠错，可以支持 2个磁盘坏还正常工作。

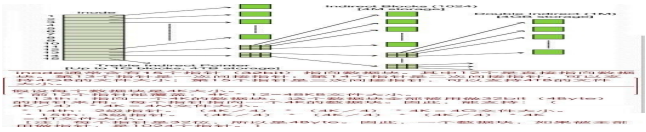


Level	Description	number of drives	Fault tolerance
RAID 0	Block-level striping without parity or mirroring	2	0 (none)
RAID 1	Mirroring without parity or striping	2	n−1 drives
RAID 2	Bit-level striping with dedicated Hamming-code parity	3	One drive
RAID 3	Byte-level striping with dedicated parity	3	One drive
RAID 4	Block-level striping with dedicated parity	3	One drive
RAID 5	Block-level striping with distributed parity	3	One drive
RAID 6	Block-level striping with double distributed parity	4	Two drives

63、**Unix – like**文件系统一个重要概念是： **"Everything is a file,except process".** 除了进程都是文件。如何把进程与文件联系起来？抽象化。通过进程文件打开表，指向系统文件打开表**open**（文件）调用返回的**id**其实就是进程文件打开表的数组下标。其中一个重要数据结构是 **inode**：每个文件对应和只对应一个 **inode**。**1:1**。通过 **inode**数据结构里的指针可以定位文件的数据块 **s**。通过**inode**的 **#**，可以在磁盘里通过读写相应的 **inode**数据。第一个 **inode**数据跟在**Boot**（ **1024Byte**），**SuperBlock**（ **1024**）后面。 **inode**数据结构里除了指向数据块的指针（ **15**个）之外，还含有许多**MetaData**控制信息。**MetaData**是文件系统常用的一个词汇。 **MetaData**就是**“Data that Describes Data”** 。描述数据的数据。这个词来源于以前图书馆里的卡片，卡片上的数据除了书的简单信息之外，有描述图书馆的书在几楼，那个书架上的索引，从而同学们可以在相应的书架上找到想阅读的书。

64、系统文件打开表是一个共享的数据结构，可以使得进程之间对一个文件（ **inode**）共享成为可能。两种方式： **1.** 父子进程之间指向同一个系统文件表的元素 **,** 共享对一个文件的操作，包括当前读写的偏移量。 **2.** 不同进程之间通过不同系统文件表表项指向同一个 **inode**（文件）。
65、**“**系统来自演化**”**。 **Linux**文件系统从最开始依托**MINIX**的简单文件系统，然后从早期 **Unxi File System**（**UFS**）开始演变，历经 **EXT**（**Extended File System** ，开始支持**VFS Interface**） ， **EXT2**，**EXT3**，**EXT4**和**BtrFS**等。不断优化。其中 **EXT3**是一重要的飞跃，支持卷（ **Journal**）。

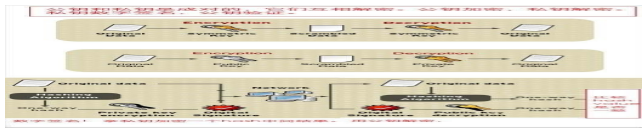
如何通过一个inode # 来定位属于中哪个块组，和在块组里是第几个inode记录：
block group = (inode − 1) /INODES_PER_GROUP
index = (inode − 1) % INODES_PER_GROUP



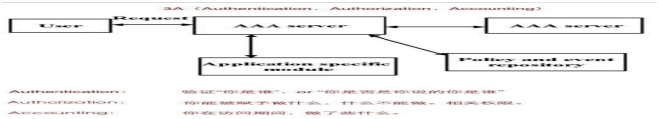
66、操作系统的安全等级规范最开始是 **NSA**发布了一个橘皮书规范**TCSEC**，定义了**5**个安全等级。 **TCSEC(1983**年－**1999**年）现在已经被 **CC**（**Common Criteria**）和 **FIPS**等安全规范逐步取代。**CC**定义了一个软件产品如果需要卖入到敏感部门所需要通过的相关需求。

67、**LSM**：**Linux Security Modules**（ **LSM**）是**Linux**领域在可信计算方面的工作，其中 **NSA**主导的**SELinux**（**Security Linux**）是基于**LSM**的一个分支，并已经在**2.6**进入主线。总体而言，安全的 **Linux**的基本框架是在进程需要获得任何核心资源的时候，需要得到相应的权限和资格检查。

68、加密算法本身是一个学科，操作系统范畴接触一些概念即可。对称和非对称加密（ **aka**，**Public Key** ）算法各有优缺点。可以单独或者混合使用，例如 **SSL**。基本流程如下：利用非对称加密算法，成功协商（交换）对称秘钥（浏览器与 **web**服务器），然后在对称算法下交换数据。



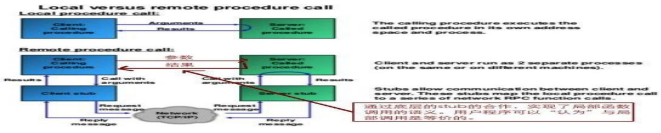
69、在操作系统的安全控制中， **3A**是关于认证（**Authentication**），授权（ **Authorization**）和记录（**Accounting**）的统一安全监管范围。**Authentication**：你是谁，允许访问否？ **Authorization**：你访问的权限和范围是什么？ **Accounting**：你访问期间，都干了些什么。



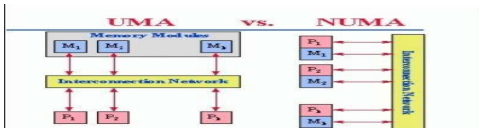
70、网络与分布式系统的界限比较模糊。共性是：都是基于松散式互联架构的拓扑，例如，基于以太网。 分布式系统比较强调资源（**Resource**）的透明性（**Transparent**），例如，一个提供的服务在哪个网络的节点上，对服务的请求者是透明的。

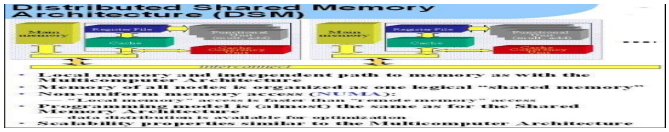
71、分布式系统有两个不能完全统一的目标： **1.**资源使用的透明性； **2.** 松散式的并行计算。过去 **20**年来谷歌等互联网公司成功的把第 **2**种模式良好的实践。而经典的追求资源透明性的系统并没有成熟和普及。 **"**牺牲透明性，追求并行性 **"**是现代分布式计算的一个折衷（ **Tradeoff**）。

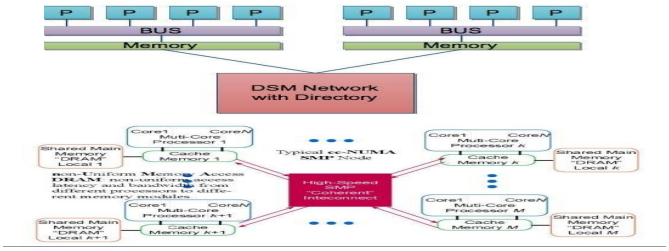
72、**RPC**为分布式系统提供了重要的局部函数调用语义，使得应用不需要过分关注网络细节，例如网络编程。 **RPC**变种有**Java/RMI**，**OMG/CORBA**和一些基于**http**承载的 **RPC,** 例如**XMLRPC**。不管通信是基于**L3**的 **TCP**，**L7** 的**HTTP**，目标都一致：局部函数调用语义从而简化分布式编程模型。



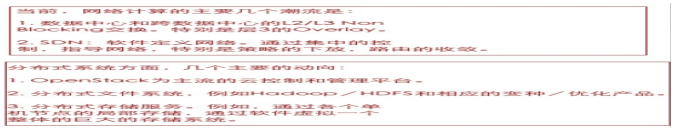
73、类似 **RPC**，**DSM** 是为了给应用程序提供一个类似局部访问内存的编程模型。通过一个数据一致性算法，保证各个 **CPU**／节点看见一个一致的内存空间。 **UMA**：多个**CPU**直接能够通过内存总线访问内存。 **NUMA**：通过互联网络（例如， **infiniband**）的拓扑。通常**NUMA**也缺省指**ccNUMA**。







74、分布式系统： **1.** 不存在一个全局的时钟。 **2:**任何一个时刻不存在一个节点，能知道那个时刻的全局系统状态。原因是：消息传递有时延。 **Lamport（2013** 年图灵奖）逻辑时钟算法精髓是：通过收发消息这天然的同步点，把无序的分布式事件，映射到整数集合上，从而自然排序。



顶 2 踩 2

- 上一篇 Android 插件化框架 DynamicLoadApk 源码解析
- 下一篇 给 Android 开发者的 RxJava 详解

我的同类文章

操作系统（1）

- 进程间通信方式 2016-06-19 阅读 121

猜你在找

- 话说linux内核-uboot和系统移植…
- CentOS7 Linux系统管理实战视频…
- 马哥Linux教程-Linux操作系统基…
- Redis内存管理和优化
- 大数据培训（第三季）——Linux…
- 嵌入式操作系统一些基本概念
- 操作系统的一些基本概念
- windows操作系统驱动的基本概念
- 操作系统的基本概念
- uCOS-II原理及应用嵌入式实时操…

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase