5.1空间的协调—主存储器（内存管理）

一，内存的分配

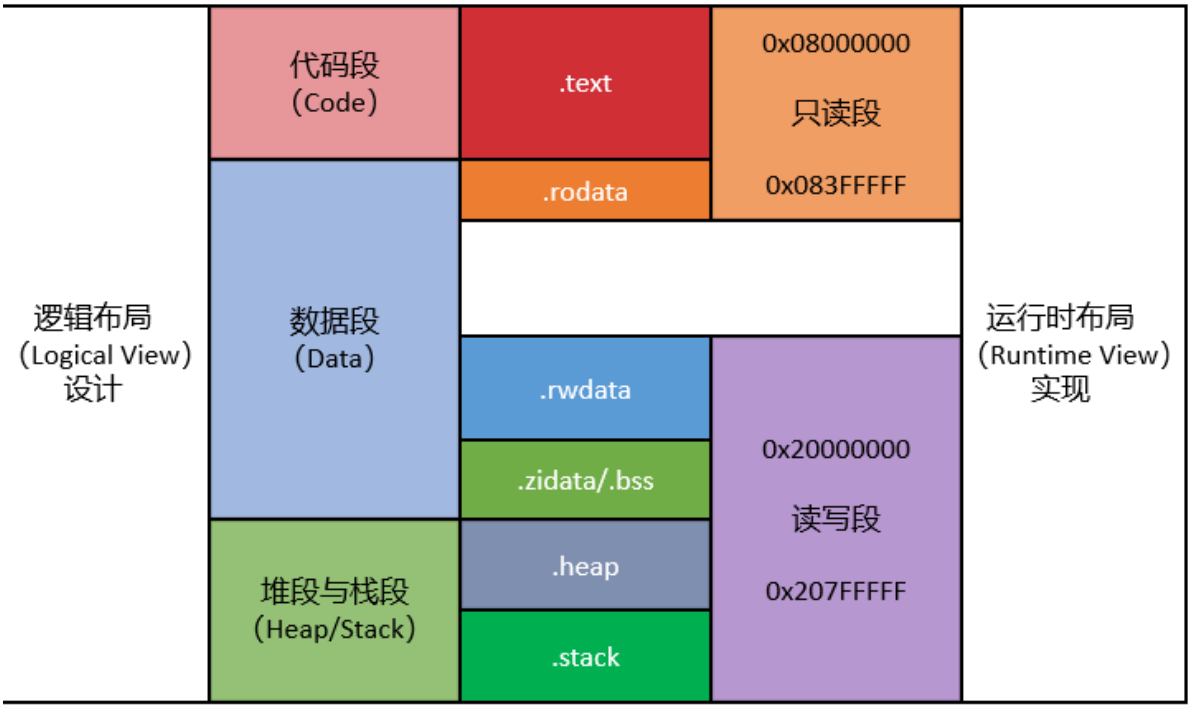
1.1程序内的分配

分段——程序再执行时的虚拟地址布局是分段的

.text/.rodata/.rwdata/.zidata——存放各个大小固定的段，分配时是已知的

.stack——分配也是已知的，而且这个段一般比较小

.heap——最大的段，程序中所有的内存分配请求都在这一段进行



应用程序内部——指令流的分工

1. 按性质分工。不同指令流处理不同性质的工作，如一些指令流主要负责I/O，另一些负责计算等。
2. 按对象分工。不同指令流处理不同部分的工作，如每个指令流负责处理一部分数据或一个服务对象

动态内存分配

原因：

1. **功能性质的不确定性**——很多应用程序有很多功能，但**用户这次要用哪个功能它不知道。**因此，它必须等待用户的决定才能加载那个功能对应的数据。当然，也可以在应用程序启动时加载一切功能，但那些用不到的功能就造成空间浪费。
2. **工作阶段的不确定性**——哪怕对于同一个功能，在**不同的工作阶段，它需要的内存的数量也是不同的。**当然，也可以直接按照最大用量阶段的内存用量进行分配，但此时就会造成空间浪费。
3. **服务对象的不确定性**——即便在同一个工作阶段，**如果该阶段针对的对象的大小不同，为了容纳这些对象，使用的内存的数量也会不同。** 当然，也可以直接按照最大的对象来分配内存，但对于较小的对象而言这将造成空间浪费。

动静态对比：相对于静态内存分配，动态内存分配最大的特点就是其动态性。动态性是为了再不确定的情况下节约内存而引入的，如果一台计算机的内存是无限大的，或者工作性质，工作阶段，服务对象是完全正确的，不需要动态内存分配。

动态内存分配请求的生命周期

分配——>使用——>释放

分配——应用程序的某个指令流发出一个**从堆中申请内存的请求**。堆的大小是已知的， 某个应用程序内部的分配算法将从堆切出一部分内存并返回给这个请求。

C语言的**malloc、calloc**等函数会从堆中申请内存。

使用——应用程序中的某个或某些指令流持有该内存块一段时间。在这段时间里，它们可能会读写该块内存。

释放——指令流不再使用这个内存块，并将其**归还回堆中**供以后申请。

C语言的**free**函数将会释放内存，将内存归还给堆。

C语言接口回顾

malloc 在堆上申请一段内存空间。

函数原型 void* malloc(size_t size)

参数 size_t size - 要分配的内存的大小。

返回值 void* - 指向新分配的内存的指针。

calloc 在堆上申请一段内存空间并清零。

语义相当于先malloc(nitems*size)大小的空间，再memset为0。

函数原型 void* calloc(size_t nitems, size_t size)

参数 size_t nitems - 要分配的对象的个数。

size_t size - 每个对象的大小。

返回值 void* - 指向新分配的内存的指针。

free 归还由malloc申请的内存空间。

函数原型 void free(void* ptr)

参数 void* ptr - 由malloc或calloc返回的指针。

返回值 无。

如何分配堆中内存?

目的：保障分配的效率，不浪费内存

1. **离线分配法**——如果直到每次请求的大小以及分配和释放的时间，原则上就可以设计除一个算法，可以用最小的堆来满足分配

问题一：假设太强，事实上不可预知

问题二：即便大小和时间已知，这个离线动态分配问题是一个NP问题（解可以在多项式时间内检验，归约到图的顶点着色问题）

这个分配方法存在的意义：是一个值得追求的理想。。。。

2. **在线算法**

竞争比——对于某种测量结果好坏的负向指标S（S越大越不利），对于任何一个输入集K中的具体输入k，如果一个在线算法ALG其输出的解的指标SALG与解决同样的问题的最优离线算法的解的指标SOPT相 比恶化最多N倍，则称该在线算法的竞争比为N。

$\forall k \in K,\ S_{ALG}(k) \leq N \times S_{OPT}(k)$

这个算法反映了某个在线算法为了在线输入的不确定性而付出的代价。一个好的在线算法拥有较低的竞争比，这意味着它深谋远虑，为未来的可能性留足了风险管理空间，因而在不确定性方面付出的代价小。

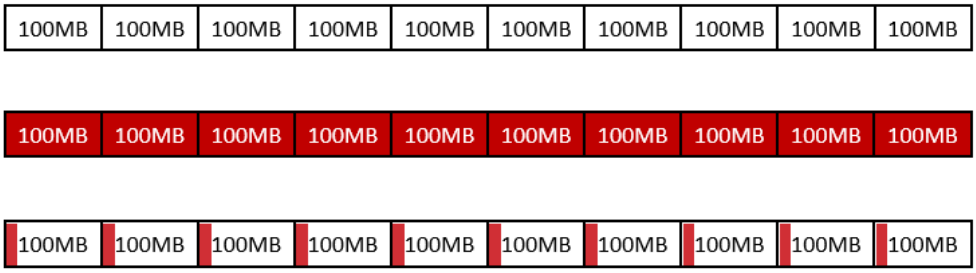
内存分配

对于内存分配问题，这个指标S可以定义为**满足某种分配的堆大小**。因此，对于同样一个分配-释放序列，算法的质量越差，需要的堆就越大，这个指标就越小。

或者说，对于需要的最小堆大小不同的内存分配算法，需要堆越多的那个算法对堆空间的利用越低效、浪费越多，竞争比也 就越差，越没有竞争力。

程序内的分配策略

固定分区法——对空间划分为多个大小相等的内存块，每次分配内存的是偶都分配固定的一块空间。（仅适合最简单的应用程序）



碎片——出于某些原因，无法有效利用而被浪费掉的资源。

在这里是出于内存分配策略被浪费掉的内存空间。

内部碎片——实际上已经指派给某个分配，但是逻辑上无法被这个分配利用的资源。这通常是由于分配粒度导致的；如果实际分配的粒度 和分配请求的粒度不一致，每次分配的资源数目要向上取整到 分配请求的粒度，造成浪费。这些因为取整而额外多出来的内 存就是内部内存碎片。

如果分配的粒度与请求的粒度是一致的，不存在内部碎片。

外部碎片—— 尚未被实际分配出去，但因为某些逻辑上的原因无法分配的资源。这通常是由于资源请求的空间分布限制与实际资源的空间分布方式冲突致的。最常见的冲突是请求的连续性与资源分布 的不连续之间的冲突，在内存分配问题上尤其如此。如果某些资源在空间上不连续，即便总量足够，也无法满足连续分配的需求。

多固定块法——在单固定块的基础上修改一下，维持多个内存块池。每次动态分配的时候，从最接近的能满足大小的内存池中分配一整块。（仅适合简单的应用程序）

动态分区法——将整个堆切成与某个粒度相同的小块，粒度由所有分配的最大公约数决定。**每次分配都分配一个或连续的多个小块。**作为一个特例，如果每个小块的大小都是1字节，那么对任何分配而言都不会产生内部碎片。

首次适配法（First-Fit）——在空闲区列表中**按某个线性顺序**（最常见的是空闲区的地址序，也可以是各个空闲区在列表中的登记顺序或者从上次分配的位置开始依次往后）**检索**，**第一个能容纳该分配的空闲区被选中**

优点：简单，对大小内存申请都公平，没有倾向性

缺点：不试着**保留大的整块区间**，也**不试图规避碎片**

最好适配法（Best-Fit）——遍历空闲区列表，选择能容纳该分配的空闲区中的**最小的那个**

优点：**试图推迟对大空闲区的分割**以便将其用于未来的整块分配，直到不得不分割它们。倾向于大块整块内存分配。

缺点：这么做等于劫小济大，会让小的空闲区碎的更厉害。一旦小的空闲 区都碎到不能再碎而无法完成分配，大的空闲区也要遭殃

最坏适配法（Worst-Fit）——遍历空闲区列表，选择空闲区中的**最大的那个。**

优点：试图**推迟难以用于任何分配的极小碎片的产生**以便保持内存对于一般分配的可用性。倾向于小块碎块内存分配。

缺点：这么做等于劫大济小，很快就不会有大块的空闲区剩下了。如果程序此时要分配大空闲区，那基本就分配不了。

内存分配策略的共性问题

1. 每次分配都遍历空闲列表。

   有多少空闲区，就要花多少时间， 这个时间是O(n)的。次次这样哪个应用程序都受不了这个时间开销， 尤其是运行得越久内存就越零碎，这个n就越大。

2. 策略单一

试图将所有内存都塞在一个内存池里面。这是非常幼稚的做法；就像线程调度那样，不同指令流的内存分配倾向也可能不同。有些指令流倾向于分配大的整块内存，有的则倾向于分配小的碎块内存。 将它们的分配放在一起的结果就是互相干扰：小块内存的分配导致大块内存因为空间不连续分配不了，大块内存的分配导致小块内存 因为空间不够分配不了。

3. 粒度过细

   在最极端的实现中，连1个字节也可以作为空闲区挂在空闲区列表 里面等待分配。这理论上当然消灭了所有内部碎片；然而，空闲区列表以及空闲区数据结构也都不是免费的。为了一个1字节的空闲区，花费数十字节的额外数据结构为它做登记，这个代价未免过于昂贵。

1.2程序间的分配

程序间分配的问题

1. 过度分配

   某些程序为了堆声明过大的空间或者不知道要声明多少空间，如果分配多了就会浪费，造成内部碎片，分配少了，就不够用。

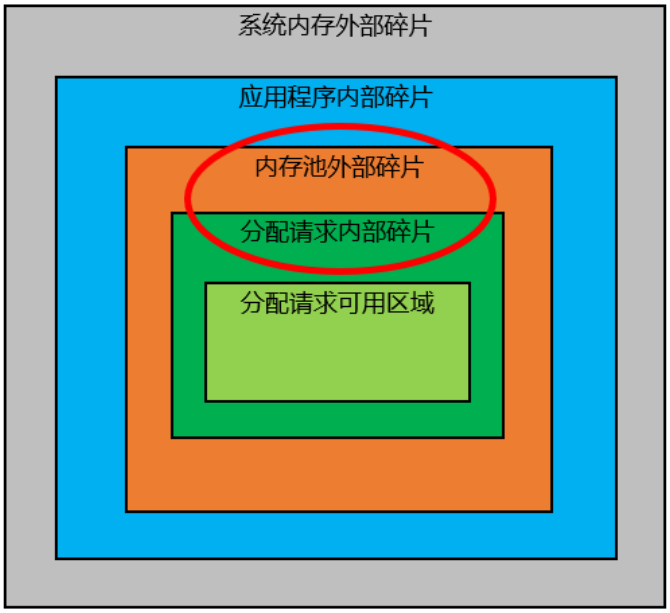
2. 内部碎片（程序内分配）

   这里指的是操 作系统已经指派给程序，但逻辑上无法被这个程序利用的内存。

3. 外部碎片（程序间分配）

   有可能出现操作系统仍有内存但因为某些原因无法分配给应用程序的情况，比如内存不连续。

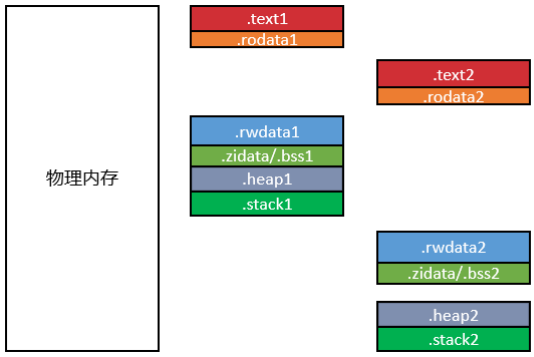
   这只有在简单分区或者段式内存管理单元上才会出现；对于页式内存管理单元这是不可能的， 除非用户请求超级页。



二，进程与内存隔离

2.1 虚拟地址

简单分区——将物理内存切割成几个块，一个块运行一个应用程序，或者放置一个应用程序的某个段。各个应用程序的各个段在链接时就决定好要放置在什么地方。



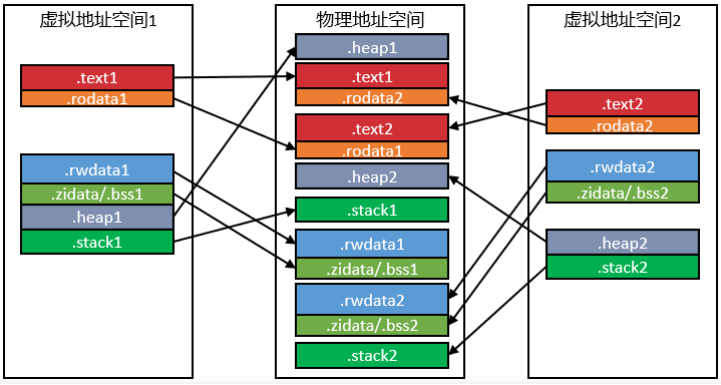
虚拟地址空间（翻译+隔离）

硬件地址翻译 采用添加额外硬件的方法，将应用程序发起的**存储器访问的地址做系统性翻译**，使得**不同应用程序中对一个地址发起的访问实际对应内存总线上的不同地址。**

虚拟地址 应用程序认为它自己在访问的地址。也叫**逻辑地址。**

物理地址 CPU实际送出到内存总线上的物理存储单元地址。也叫**实地址。**

XX地址空间 由XX地址组成的地址集合就叫做XX地址空间。



2.2进程的描述

进程Process

操作系统提供给应用程序的一种对**地址空间**的抽象机制。**它是内存分配的基本对象。**应用程序通过将自己装入地址空间与进程对应起来，使自己在内存中拥有一个活动副本。**操作系统通过给进程分配内存，来给依附在这个进程上的应用程序提供运行空间。**

也可以说，**程序通过依附于进程，获得了占用内存空间的权利。**

地址空间和进程

就和**指令流与线程**的关系类似，操作系统并不直接知道某个应用程序的存在，它只能看到某些程序通过某些方法向某个地址空间加载数据并在那里执行。只要用户愿意，且应用程序的设计许可，用户可以在一个地址空间中装入多个应用程序，也可以将一个应用程序分成多个互相协作的地址空间。

描述符标：进程本身是一个地址空间，所以必然有一个参数描述这个地址空间，它一般是一个表格，足以令操作系统决定哪些内存该进程有权访问，以及怎么访问

注：地址空间未必是虚拟地址空间。在经由物理地址空间的处理器上，如果具备内存保护单元也是可以实现进程的，因为他能实现隔离，但是这样就无法实现地址翻译了，需要自行解决应用程序之间的地址冲突问题

进程还可以包含一些其它权限描述，比如对文件、设备等的访问权限。

进程和线程

线程通过在进程里运行，将CPU时间转化为对地址空间的访问操作次序，从而实现应用程序的功能。

更本质的定义

进程是一种**特殊的保护域**，其特殊在它是常见保护域中最小的一种且经常与地址空间联动。

（**保护域（Protection Domain）**是一组**权能（Capability）**的集合，又称为**权能空间（Capability Space）**。这些权能一方面**赋予（Grant）** 进程合法资源操作的权限，另一方面**限制（Confine）**进程非法操作 资源的企图，最终实现保护域之间的**隔离（Isolation）**，

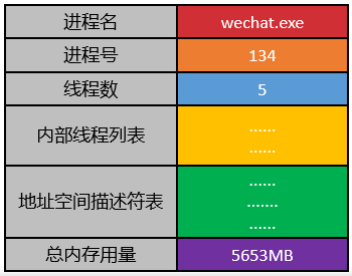
权能 一种代表对某种资源做某种操作的许可的不可伪造（Unforgeable） 的令牌（Token），保护域获得它的唯一方式是等待管理者为其下发。容器，虚拟机都可以看作保护域）

进程持有的地址空间描述符表可以看做是对**地址空间的访问权能**， 其它权限则可以看做是对其他资源的访问权能。不管如何，这些权能都只能由更高层次的管理者（如操作系统或其它高权限进程）下发。

进程控制块PCB

操作系统用以描述和管理进程的内核对象，一般至少包含进程的**地址空间描述符表及一些其他权限表**，有时还会包含一些**身份信息（如进程名、进程号）、统计信息（如当前正在运行的线程数、总计内存大小）、线程信息（当前的线程列表，内含指向各个TCB的指针）**等。它在一般是C语言的一个结构体。

进程控制块总是**位于内核空间**，只有操作系统可以更改，应用程序无法更改。与TCB不同，没有内核模式的CPU不需要也无法 （用硬件手段）实现PCB



进程与可执行文件

1. 可执行文件——应用程序在外存上的存储方式。它描述了应该为应用程序建立一个什么样的进程，进程中要有什么样线程，以及线程和具体的指令如何对应。它是死的，干瘪的，静态的应用程序，没有执行环境和上下文，内有执行活动。
2. 进程——应用程序在内存中的活动组织，它是活的，丰满的动态的应用程序，具备一个由地址空间和其他权限提供的执行环境，并充满了线程的执行活动和上下文
3. 关系——可执行文件对进程为一对多关系。一个可执行文件每启动一次就可以创建一个（这是通常的实现）或一组进程；如果它启动多次，就可以创建一系列或一系列组进程。

同一个可执行文件，在启动为不同的进程时，可以处理不同的工作、 使用不同的权限，或者以不同用户的名义启动。生成的多个进程之间是不同的，因为他们内部的执行环境、执行活动和内部线程的上下文均有差别。

进程和线程

线程通过在进程内运行，将CPU时间转化为地址空间的一个访问操作次序

线程——CPU执行时间的分配对象，指令流通过依附于它获得执行时间。但它又需要依附在进程上获得执行空间

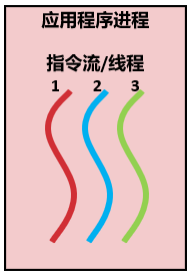
进程——仅仅一个执行空间，本身不具备执行能力。作为特例，一个进程在创建时可以不包含线程，而是等待其他进程中的线程迁移过来。

关系——一对一、一对多、多对一、多对多关系，它们在数量上没有任何固定的对应关系

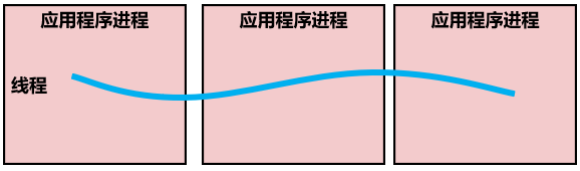
一对一关系：最常见关系，在Linux2.4前，线程和进程是一个东西，task_struct里面包含线程的信息和地址空间的信息。

一对多关系：常见的关系，一个进程中同时存在几个线程，多线程进程，

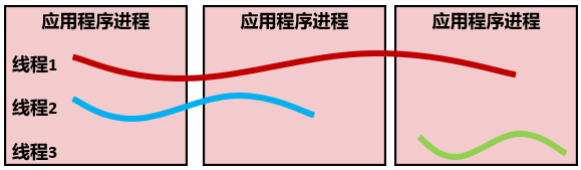
本质是多个CPU时间分配对象共享一个内存空间分配对象



多对一关系：罕见的关系，线程可以被称为迁移线程，它会在多个进程之间（以受控的方式）游走，在不同的执行阶段使用不同的地址空间和权限，但始终使用同一份CPU时间预算。



多对多关系：



2.3内存隔离机制

内存隔离机制：**分段**，按段划分虚拟地址

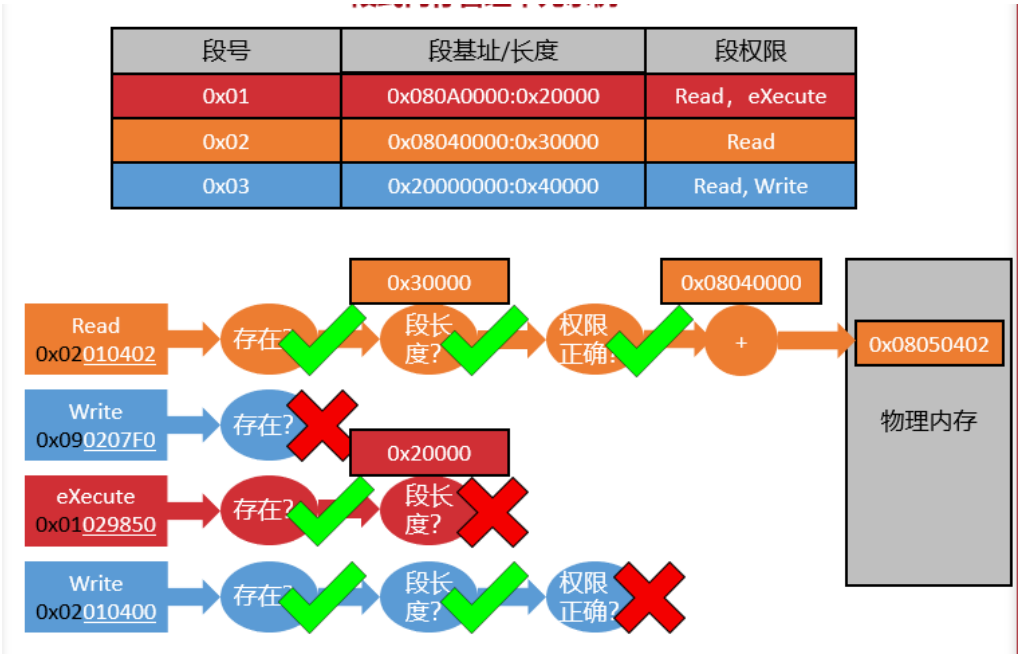
段式内存管理单元（S-MMU）

常用于分段布局的存储器访问管理工具，具备**按段地址重映射**和**访问权限管理**两个职能。

段式内存管理单元使用一张**段表**，每段都包括“**段号**”，“**段物理地址范围**”和“**段权限**”三个部分、

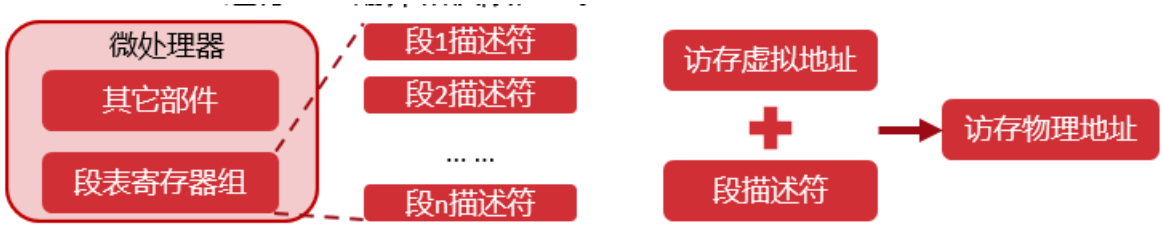
应用程序访问发起每一次内存访问都需要经过段表的转换和检查

1. 按照访问的拘泥地址中的段号信息查找相应的段
2. 发起的访问的性质必须式该段的权限允许的
3. 访问的物理地址=段基址+虚拟地址，且虚拟地址不得超过段长度



段表的查询

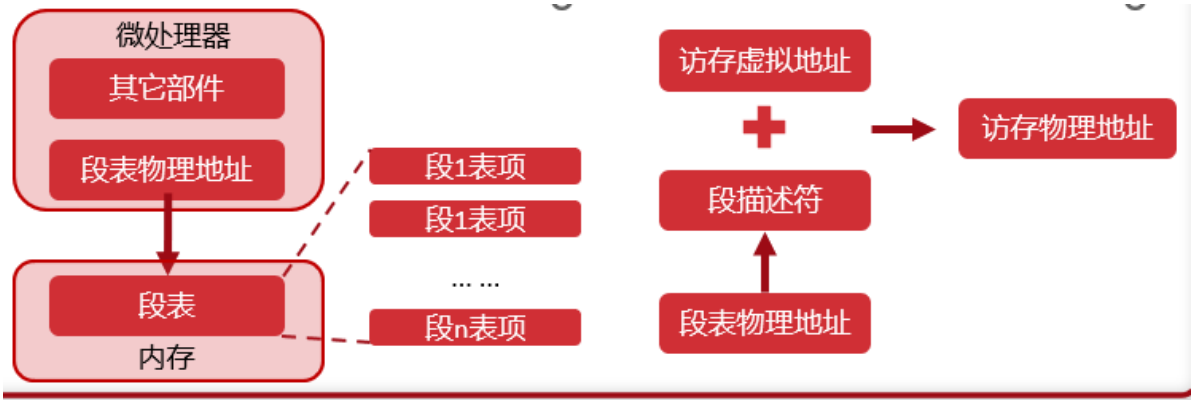
段表放在CPU专用寄存器，进程切换时，需要从PCB取出新进程的段表，然后拷贝到CPU专用寄存器中。这种是简单的情况，很多是一系列寄存器组



优点：段的描述是存放在CPU寄存器中的。CPU在生成物理地址时只要查询这些内部寄存器就可以，无序额外访问

缺点：无法直接支持比CPU的段寄存器组数量还多的段，因为段寄存器的数量时在设计CPU时就决定了的。

解决方案：将段表存储在内存中。每次内存访问，需要先查询在内存中放置的段表，然后再根据该表项的内容计算真实的物理地址。CPU只提供寄存器来指向段表在内存中的物理地址。



优点：能支持无限目的段，段表放在内存中，大小几乎是无限的

缺点：每次访存都膨胀成两次，首先访问段表，然后再访问内存本身，现代CPU的访存延迟都很高，这势必造成严重的性能损失，尤其当段表也不在数据缓存中的时候。

问题：如果即要享受访存效率，有希望得到足够多的段?

提示：缓存以及其工作原理

局部性：

一个活动操作的对象具备某种关联性。在这里是指，一个指令流（活动）在一段时间内访问的存储器总是有某些集中性 （关联性的一种体现方式），那些集中性称为局部（短期调度的工作集）。它还可以细分为时间局部性和空间局部性。

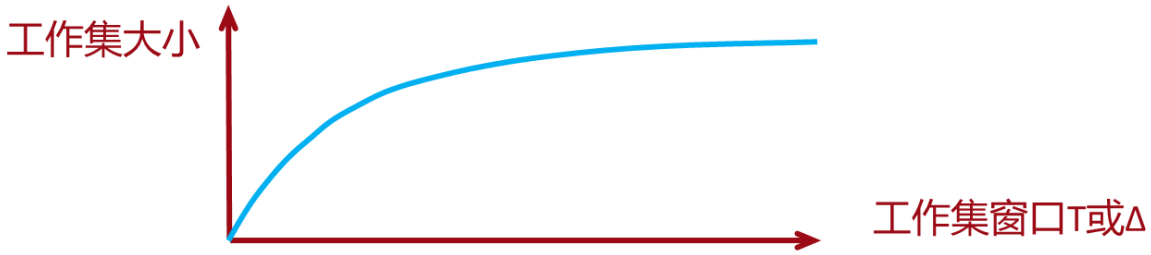
1. 时间局部性，如果某存储单元被访问，那么在近期它很可能还会被再次访问。
2. 空间局部性，如果指令流访问某存储单元，那么近期内很可能会访问附近的其他存储单元。

工作集

再某段时间间隔里，进程实际访问页面的集合

工作集——进程在某个时刻t之前一段时间间隔T内访问内存的地址的集合，即为WS（t,T），T成为工作集的窗口，用访问次数Δ代替时间间隔T，

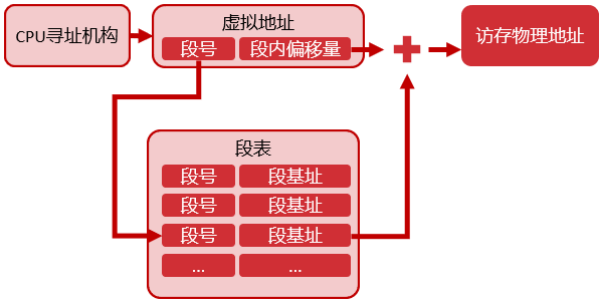
工作集的大小可能小于窗口尺寸

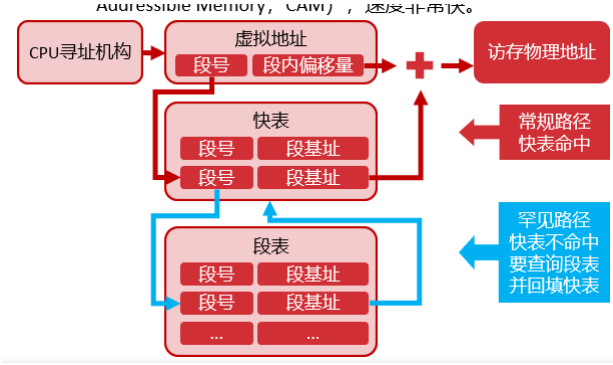


解决方案：只要将进程当前工作集包含的那些段的描述符缓存进CPU中就可以 了。这样，我们便需要一片特殊的缓存空间，专门存放这些段描述 符，还需要某种机制来填充这些段描述符。

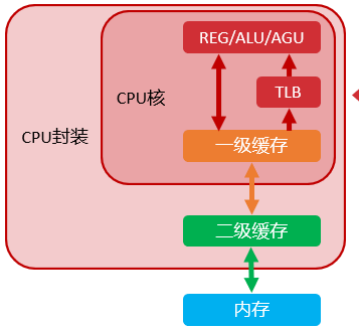
快表TLB

专用以缓存那些当前工作集中常用的段描述符，以加快地址翻译和权限检查。它也是一种缓存（Cache），并且在电路上和常规缓存一样，都采用了可并行查找的内容寻址存储器（Content Addressible Memory，CAM），速度非常快。





每次访存都要查询TLB，因此TLB和一级缓存一样，都是相当热的部件。降低它的功耗，提高它的利用率至关重要



快表填充 TLB FIlling

软件填充

使用场合：那些CPU提供一组寄存器存放段描述符的场合

当CPU在这组 寄存器中找不到段描述符的时候，便会生成一个内存管理异常，操 作系统可以截获并处理该异常。在处理流程中，操作系统可读取段 表，然后将段描述符手动填充到CPU的段寄存器组中。

这种方法非常讨巧，等同于把段寄存器组变成了快表。

优点：段表是软件查询的，因此什么格式软件说了算，而且操作系统在任 何时候都能控制段表的内容，添加段描述符到段表中时也可自行决 定替换哪个段。

缺点：内存管理异常非常常见，而软件处理非常消耗CPU时间。另外，如 果段表很大，切换进程时拷贝整个段表到寄存器太慢了。

硬件填充

适用于那些CPU提供真正的TLB来存放描述符的场合。

当CPU在TLB 中找不到段描述符（不命中）的时候，它会自动调用表查找器（Table Walker） 硬件，该查找器将读取段表并将段描述符自动填充到TLB中。

优点：填充速度快，平均性能好。

缺点：段表的格式由硬件决定了，而且操作系统开发者失去了对TLB内容 的控制，因为硬件会自主决定新的段描述符覆盖哪个旧描述符。

适用场合

在那些需要对TLB施加完全控制，或需要节省成本的场合，选择软件；

在那些不关心成本且在乎平均性能的场合，则选择硬件。

硬件表查找器是一个大众选择，因为它的成本相较于其它硬件 微不足道，而且填充比软件快得多。

缺段异常

指的是段表中无法找到段的情况。

软件填充情况下，内存管理异常并不绝对等于缺段异常

硬件填充情况下，任何的内存管理异常都是缺段异常

快表冲刷

快表击落

快表锁定

替换方法

快表的大小

空间和工作集

分段的问题：

产生外部碎片，不会产生内部碎片

段的连续性：存储区域无论在虚拟空间还是物理空间都是连续的，这一位置，一个段必须适用一整块物理内存。

分页

按页划分虚拟地址

将虚拟地址切割成同样大小的页，每个虚拟页映射到不同的物理页。每个程序的虚拟页到物理页的映射组成页表

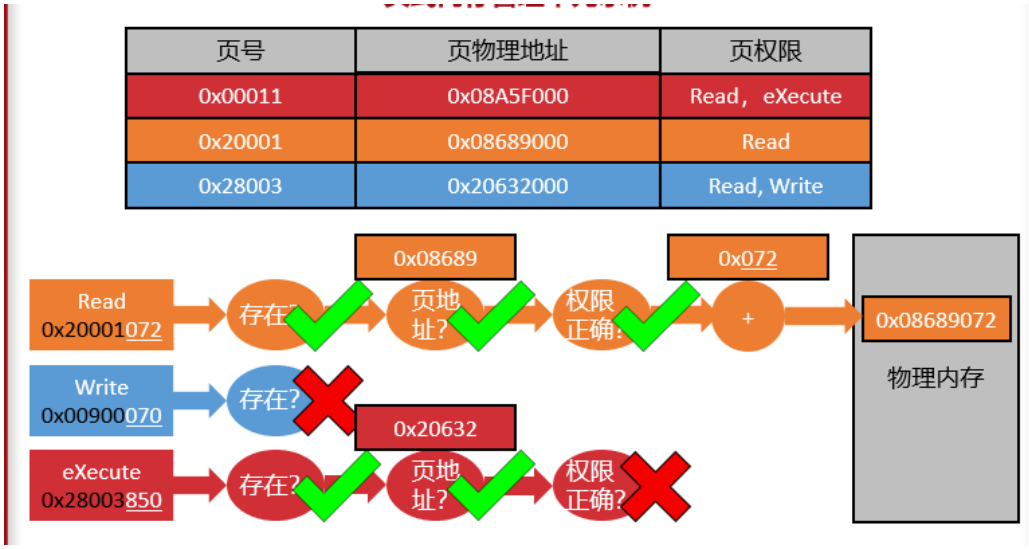
页式内存管理单元（P_MMU)

常用于分段布局布局的存储器访问管理工具，具备按页地址重映射和访问权限管理两个职能。

P_MMU使用一张页表，包括：“页号”，“页物理地址”，“页权限”

每次内存访问：

1. 按照访问的虚拟地址中的页号信息查找相应的页
2. 发起的访问的性质必须是该页的权限允许的
3. 访问的物理地址=页基址+页内偏移量



页表

1. 存放位置：相比较段表，页表很大，全部放寄存器放不下，索引只好提供一小部分寄存器和TLB
2. 存储形式：超长的线性表。

页表的稀疏性

线性表的稀疏表示

链表表示法：页穿串成链表，查找很慢，

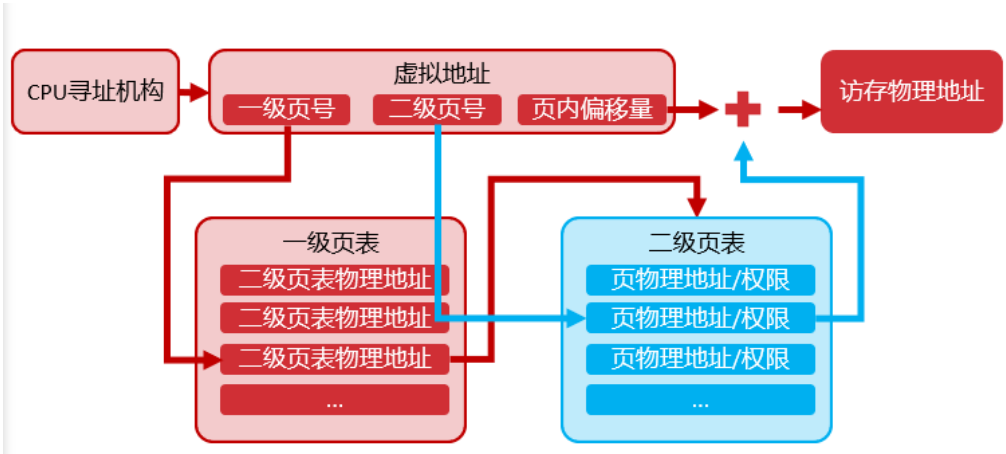
堆表示法：常规二叉查找树O（logn）

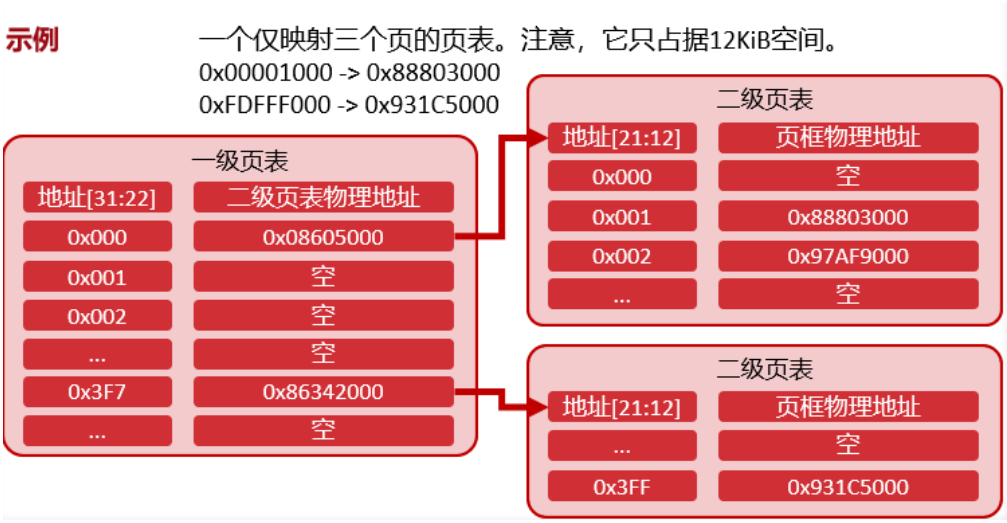
桶排序：

基数树：

以某个基数组织的“桶套桶”序列（更专业的叫法是前缀树），该基数决定了每一层桶的多少。对于页表而言，每层的桶都是2的次 方个，如1024个。

多级页表





分页的特点：

1. 不存在外部碎片
2. 存在内部碎片
3. 页越大，产生的内部碎片越多，TLB的利用效率却越高，发生TLB不命中的概率降低，

超级页：支持映射一个巨大的页而非映射一个下级页表，

快表相关：分段那里知识适用于分页

压缩基数树：页表的表示还可以继续精简：如果某级页表中只有一个小页被 映射，那么它可以直接被省去，并将这个页的信息（需包括其 大小和其所属的超级页中的偏移量）直接置于上级页表中，这 样就可以直接从上级页表定位这个页。这在那些空间紧张的嵌入式系统中非常有用。

基数的选择：页表可以选择任何基数来实现。但是，2的次方是最方便的，因 为可以直接截取地址中的某一部分来做变换。此外，在设计硬件时，页表的每一级最好都要一样大，且该大小最好要和系统 中最小的页大小一致，这样每一个页都可以用来存放一个完整 的页表级别，程序编写起来比较方便。

三，内存的活用

3.1请求分页

进程的工作集

和指令流一样，一个进程内的所有指令流的活动在一段时间内访问的存储器也总是由某些集中性。

这就意味着只要将包换进程工作集的那些页面调入内存就行

请求分页

进程活动时，将当前工作集调入内存，其他部分放在外存，之傲工作集再次包含它们。

内存被当成外存的缓存，外存充当内存的后备

这种机制叫页面交换，虚拟内存，分页文件

单层存储模型

存储器被抽象成一种逻辑模型。

操作系统实际决定哪些耐热放在哪些层次的存储器中。

缺页异常处理

1. 缺页异常
2. 缺页异常处理，进入OS
3. 写回换出页
4. 读入换入页
5. 修改页表
6. 重新执行，PC不变

3.2替换算法

如何选择换出的页？缺页率，提高内存的带宽利用率，让内存的内容尽量紧密追随进程的工作集

最长前向距离算法LFD OPT最优替换算法

每次替换时，都寻找当前页面中在最远未来才会再次适用的那个页面，替换他

优缺点：缺页率最小，性能最好，但是无法实现

先进先出FIFO

优先淘汰最先进入的内存页面

优缺点：实现简单，但是性能很差

会出现**Belady异常**

在某些资源分配策略下，增加资源总量反而导致性能下降和效率降低的现象

这里指，对于替换算法，允许的物理页数目越多，缺页率反而升高。

最久未用法LRU

优先淘汰最近最久没有访问的也秒

优缺点：性能很好，但是需要硬件支持，算法开销大

最不频繁适用LFU

选择那些某个时间段内访问次数最少的页替换掉

随机Random

随机选择一个页替换

替换算法本身是拿时间换空间

现代计算机中会将外存的东西都放进内存，用空间换时间

能运行程序时计算机的最低目标，通常操作系统会分配多得多的页来覆盖工作集，保证程序能高教运行。

抖动（Thrashing）

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为成为**抖动**，

产生的主要原因：进程频繁访问的页面数目高于可用的物理块数（分配个进程的物理块不够）

当被分配的页数小于当前工作集的时候，缺页率会大幅增长。此时，程序的访存性能向外存的性能急剧跌落。工作集有短期、中期和长期三个层面， 抖动也是这样，存在着短期抖动、中期抖动和长期抖动。

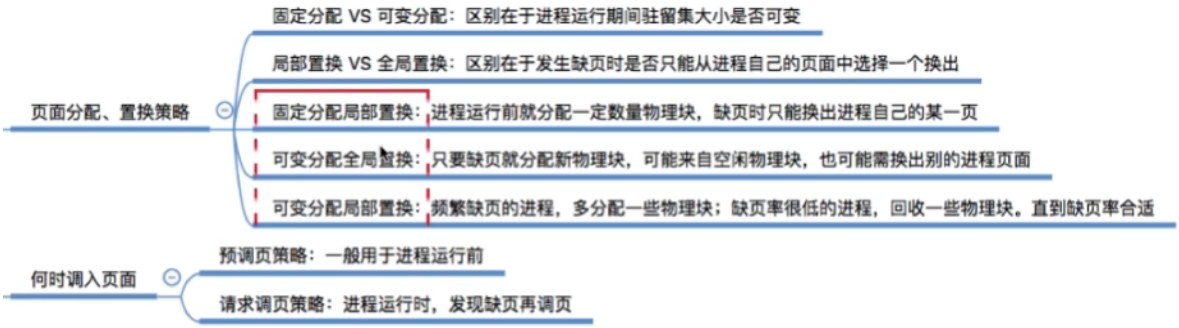
分配策略的动态性

一个进程的页的数量可以说静态设置的也可以是动态调整的。

如果一个工作集的大小已知，就分配固定值，不确定的话就阶段性的动态决定进程需要多少页框。

置换策略的全局性

页置换可以在一个进程内部发生，也可以在进程间发生。



3.3内存映射

内存映射文件

操作系统向上层程序员提供的功能（系统调用）

- 方便程序员访问文件数据
- 方便多个进程共享同一个文件

特性：

1. 进程可以适用系统调用，请求操作系统将文件映射到进程的虚拟空间
2. 以访问内存的方式读写文件
3. 进程关闭文件时，存储子系统负责将文件数据写回磁盘，并解除内存映射
4. 多个进程可以映射同一个文件，方便共享

优点：

1. 程序员编程更加简单，已建立映射的文件，只需要按访问内存的方式读写即可
2. 文件数据的读入/写出完全由操作系统负责，I/O效率可以由操作系统负责优化
3. 简化了文件的读写操作，提高了文件访问效率。

内存映射设备

类似内存映射文件

像访问内存一样来处理设备的输入输出操作

5.2 空间的协调—进程实现（主存储器分配机制和实现）

一.进程的实现

1.1系统进程总览

分类：

按进程的来源分类：

1. 系统守护进程——操作系统开发商提供的系统服务进程，不在内核，靠近内核的功能。一旦发生故障等同内核故障。可以看作不在内核的内核组件
2. 服务守护进程——应用程序或驱动程序开发商提供的功能服务程序，需要常驻后台的功能。
3. 应用程序进程——应用程序开发商提供的应用程序进程。

守护进程——提供一定背景服务功能的特殊后台进程，无前台界面

按特权划分：

1. 特权进程——有一定特权的进程，可以调用系统的特权API，访问敏感资源的特权
2. 普通进程——没有特权

辨析：特权进程vs内核进程

前者中所含的线程 是运行在用户模式下的，只不过它们拥有调用敏感资源的特权， 而且不同特权进程的特权大小可以不同；后者则是直接运行在 内核模式下，不属于任何一个进程，而且总是具备最大特权

进程的来源	特权模式	特权	运行线程	提供方
（内核） 不属于任何进程	内核模式	极高	输入输出线程 系统服务线程 空闲线程 （内核线程）	操作系统 驱动程序
系统守护进程	用户模式	高	输入输出线程 系统服务线程 （用户线程）	操作系统 驱动程序
服务守护进程	用户模式	中，低	后台服务线程 （用户线程）	驱动程序 应用程序
应用程序进程	用户模式	低	前台程序线程 （用户线程）	应用程序

1.2进程的实现

进程控制块PCB

操作系统用来描述和管理进程的内核堆笑

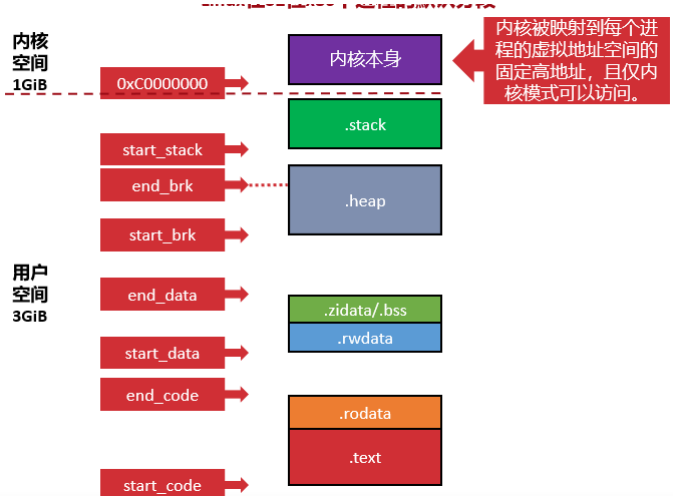
包含进程的地址空间描述附表以及一些其他的权限表，

包含身份信息（进程名，进程号），统计信息（线程数，内存大小）线程信息（线程列表，指向各个TCB的指针）等

PCB和TCB

相当多数操作系统也是这样设计的：切换线程时需要判断下一个线程和当前线程是否在同一个进程内，允许根据TCB反查PCB比在PCB中存储所属TCB的指针更加方便。当然，也可以两边都存储指针，这样可以根据PCB查找TCB，也可以根据TCB反查PCB。

内存分段



二.内存分配的实现

2.1 x86-64内存管理

段页式管理

将段式内存管理和页式内存管理结合起来使用

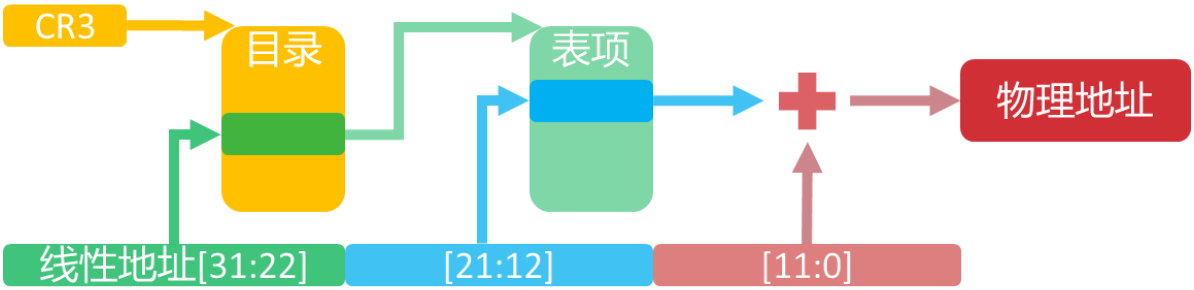
物理地址先经过段式内存管理转换成线性地址（中间地址）然后通过页式内存管理转换为最终的物理地址

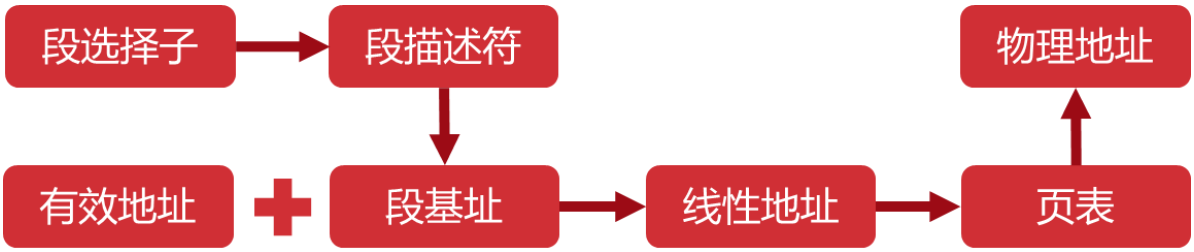
（先分段，后分页）

优越性：

1. 避免段式管理的外部碎片
2. 避免页式管理的内部碎片

	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片



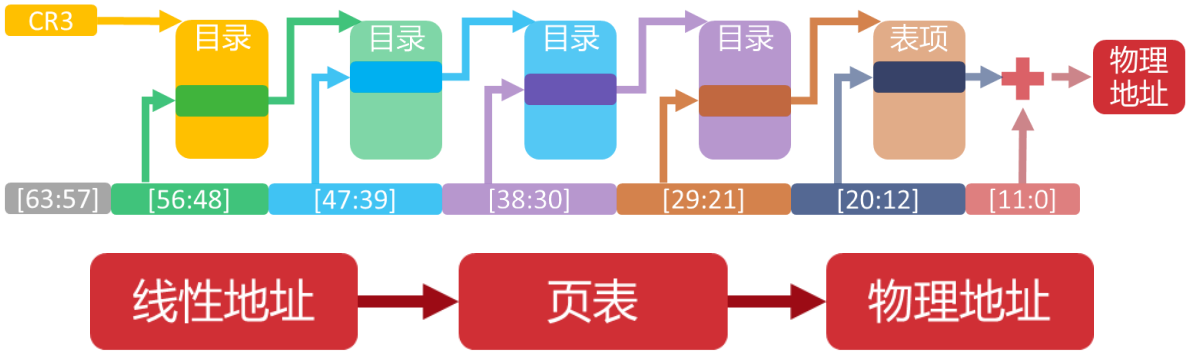


长模式分页

将CPU设置为64位工作模式，必须先进入保护模式，然后再切换到长模式

长模式分页

段式内存管理被基本禁用



问：为什么【63，57】没翻译，只翻译后面？

x86-64规定，虚拟地址空间是不连续的，分成内核空间，空白区域和用户空间，不落在这两个范围内的地址都是不规范的非法地址，不查询页表， 直接按错误处理。实践中，我们一般让内核使用高地址，用户程序使用低地址。

x86-64分页机制详解

CR0

CR3

CR4

PML5

PML4

PDPT

超级页

页式内存管理中，页目录允许直接映射一个巨大的页，这个页的大小一般是下级页目录的全部大小。

这使得系统中可以穿线多种页大小，进一步节约了系统页表存储空格键和翻译空间。节省了TLB条目

对于因公程序而言，分配超级页能大大提升他们的性能，因为同样的快表条目数量能覆盖大得多的内存。

代价：

失去翻译的灵活性，一个超级页内部的地址无论是虚拟地址还是物理地址都必须是连贯的。如果一个应用程序非超级页不用，当系统中只剩下的小页面时就无法给它分配了。零位硬件页必须调整来使用这一变化，增加成本。

TLB清空的问题：

在操作系统的语境下，内核会被映射到每个进程的高地址（映射为仅内核模式可访问），且和进程共享TLB。每次进程切换时，TLB都需要被冲刷，那么属于内核的条目就也被清空了。

不过，因为内核映射到每个进程的高地址的方法是一致的，实际上没有这样做的必要；在不同进程之间，虽然进程的页面映射和工作集会改变，但内核的页面映射和工作集并不会变化。

全局页（global Page）

为了在切换进程时不清除那些输入内核的页面的TLB条目，现代处理器都加入了全局页的机制。

首先，在映射内核的页面时，一律将G位置位，将它们映射为全局页面，这样他们的条目就不会在TLB冲刷是被清空，TLB冲刷仅仅清理那些非全局页的条目，内核的条目则得到跨进程保留，提高了内核运行的效率。

全局页机制，可以看作一种优化，他使内核的工作集核应用程序的在同一个地址空间内的工作集得以分离，并在需要清理其中一个时不干扰另一个。

PGD/PT

地址空间编号

跨进程工作集

每次进程切换都会彻底清除TLB的内容，这就不可避免的对性能造成负面影响。

工作集共存 如果能够在切换进程时，保留上一个进程的TLB内容就可以让它们的 工作集共存了。但是，简单地这样做会导致两个进程的条目在TLB中 同时存在，从而破坏进程的边界。如果能够发明一种机制，让仅属 于当前进程的TLB条目发挥作用，而其它进程的TLB条目不发挥作用 就好了。

地址空间编号 一种能使不同进程的TLB条目共存，但仅使当前进程的条目生效的手段。不同的进程被赋予了不同的编号，在切换其页表时被一起装入CR3寄存器。在TLB条目填充时，该编号被一起填充进TLB；同时，只有编号与CR3中的编号相同的TLB条目才起效，其余条目视为无效。

进程内空间分配的指导原则

高效 应当节约内存空间，少产生内外碎片。

快速 应当很快完成分配，少浪费CPU时间。

问题 除去这两点之外，还有什么别的标准?

可移植 可以在很多操作系统/软件环境中上移植和使用。

可配置 参数应当可以调整以适应具体的部署环境。

局部性 尽量将那些一起使用的内存段分配到靠近的地址， 以减小TLB缺失和缺页。

错误探测 尽量能够探测到内存错误。

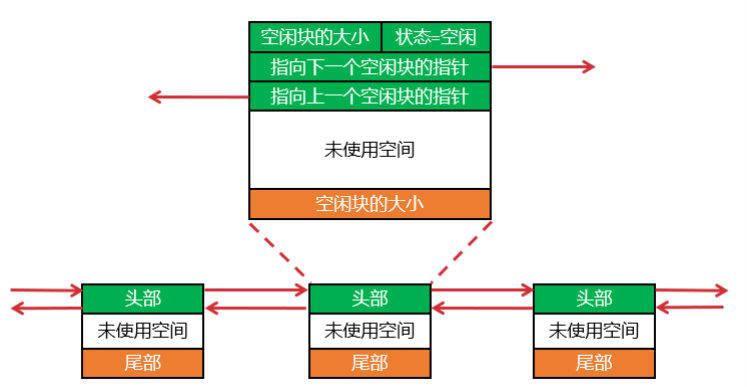
结论 对高效和快速的追求决定了我们必须使用动态分区法。但是，简单 的策略是必然无法满足以上追求的。

2.2DLM进程内分配

dlmalloc malloc在Unix等系统中的默认实现。由Doug Lea于1987年发表， 因此得名。

性质 一种动态分区分配器，在很多场景下有不错的空间性能和时间 性能，并能在很多其它次要性质上有不错的表现。

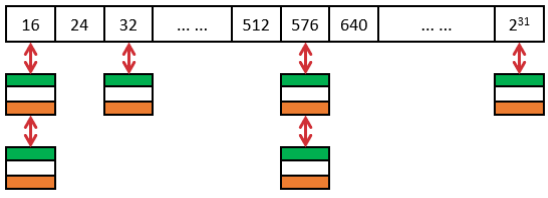
dlmalloc对空闲内存块的表示



问题一 为何使用双向循环链表?

问题二 空闲块的大小为何在首尾各出现一次？尾部那次可以不要吗?

dlmalloc对空闲内存块的组织



最小粒度 dlmalloc规定，一个内存块最小是16字节。如果小于16字节，就 不再切割，放任它们成为内部碎片。

块队列 dlmalloc将空闲块按照其大小分成127个队列，从最小的16一直到 最大的231。16-512号队列按照8字节的等间距展开，且每个队列 中空闲块的大小必须等于队列号。从576号队列开始，队列按照 指数间距展开，且每个队列中空闲块的大小都必须大于本队列 号，但小于下一个队列号；内存块按照从小到大登记在队列中。

问题一 为何放任比最小粒度小的内存块成为内部碎片?

问题二 为何要设计多个队列，且小队列和大队列的规则不同?

伙伴分配器（Buddy Allocator）

伙伴分配器 按照一系列不同大小的队列组织空闲块的分配器。它一般采取 最好适配法进行内存分配。

内存块的拆分 当分配时，从可能满足分配的队列（要分配610字节，需要查询 哪个队列？）开始查询合适的空闲块，一直到找到该空闲块为 止。对该空闲块进行拆分，分配合适的大小，再将剩余部分挂 回合适的队列。

内存块的合并 当释放时，查询空闲块前后两个内存块的状态。只要其中任何 一个也是空闲的，就将它们合并进这个空闲块，然后将这个大 空闲块挂回合适的队列。

问题一 伙伴分配器的时间复杂度是多少（考虑分配和释放）？

分配 找到合适的内存块是O(logk)，切割是O(1)，把内存块插 回512以内的队列也是O(1)，把内存块插回合适的队列是 O(logk)，k是队列的平均长度。

释放 合并是O(1)，把内存块插回合适的队列是O(logk)。

总之，伙伴分配器的时间复杂度是O(logk)。虽然k与内存块总数n 成正比，但由于队列很多，k一般很小，总的时间复杂度在实践 中接近O(1)。不过，伙伴分配器在极限场合仍是可以恶化到 O(logn)的；关于严格O(1)的TLSF分配器，我们留到后面讲解。

优化 在分配时，伙伴分配器需要从最小可能的队列开始，一个个查 询队列中是否有空闲块。由于队列的数目是有限的，原则上讲 这个操作仍然是O(1)的。但绝大多数内存分配都需要经过这个过 程，因此在实践中它需要被最优化。我们往往会维持一个位图， 其中的每一位都对应一个队列。当队列中有块时，这个位置位， 没有块时这个位清零。这样，只要查询该位图就能够确定对应 的队列中是否有空闲块了。

问题二 我们都知道最优适配法会导致大量的小内存碎片。为何dlmalloc 不担心这一点?

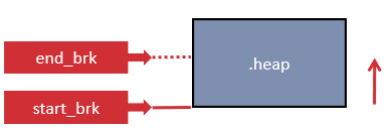
最小大小 低于某个最小大小的空闲块将作为内部碎片， 因为为它们创建任何管理数据结构都是不合算的。这也是为何动态分区法在实践中有内部碎 片的根本原因。

专用队列 小队列是专门为频繁发生的小内存请求设计的， 它们的大小是确切大小。这一来保障了小内存 分配的时间复杂度，二来保障了不会产生任何 无法利用的外部碎片。如果某分配切分后的空 闲块不正好是某个小队列的大小，那么就干脆 多切给该分配一些，直到空闲块等于某个队列 的大小为止。多切的部分就作为内部碎片。

问题三 小队列的间距选择和最小队列的大小选择需要权衡什么因素?

问题四 考虑到内存分配器的这种特性，我们写程序需要注意什么?

向系统要更多页面



程序间隔 前面讲过，在Unix系统中，堆段并非一次分配出去的，否则就 浪费内存。实际使用的是一种“程序间隔”机制：系统认为数据段和栈段之间的“间隔”全都是堆段，其中映射了内存的部分记载在start_brk和end_brk两个变量中。当进程内分配器将两者间内存 用完，便要请求操作系统向上移动end_brk，从而将更多内存存加 入堆段。这使用sbrk系统调用完成。一旦end_brk上移，一般就 不会下移回来了（为什么？），因此不到万不得已是不会上移 end_brk的。

sbrk 扩张堆的实际大小，从操作系统请求更多内存。

函数原型 void* sbrk(intptr_t increment)

参数 intptr_t increment - 向上移动的字节数。

返回值 void* - 指向原end_brk的指针。

巴拉巴拉太多了

2.3Linux进程间分配

巴拉巴拉太多了

看ppt去吧

真的多，而且看不懂

2.4Linux页面替换

原则 提高内存的利用率，满足尽量多程序的内存分配，最大程度减 少缺页异常的数量。

理想策略 如果能实现FIFO、LRU或者LFU就好了。当然，LRU和LFU又比FIFO 好。如果都不行，就只能实现RANDOM了。

硬件机制

1. **缺页异常** 当页表中没有某个页时，产生操作系统可截获 并处理的异常。
2. **访问位A** 当某个页被访问，其页表的访问位会被硬件置位。
3. **脏位D** 当某个页被实际写入（内容遭修改），其页表的脏位会被硬件置位。

问题 用这些硬件机制怎么实现FIFO、LRU或LFU?

LFU 不太好实现，因为硬件没给我们提供访问频次计数。如果要实现的话，就必须在每次访问内存时候软件介入注明自己访问哪些位置，这是不现实的。

LRU 也不太好实现。LRU要求我们给每个页面提供一个计时器，记录其上一次访问以来经过了多少时间。硬件上没给我们提供这种计时器；考虑到页面的数量，软件实现也不现实。

更要命的是，LFU和LRU都要求在每次页面替换时扫描所有页面， 然后找到那个次最小或者访问经过时间最长的页面。这就更 不现实了。

问题 LFU和LRU策略因为缺乏相应的机制，都实现不了。那只能求助 于FIFO策略了。FIFO策略需要哪些硬件机制？它好实现吗?

FIFO FIFO肯定是能实现的。把页面组织成一个链表或者环形缓冲区， 然后先进先出就可以，效率很高。

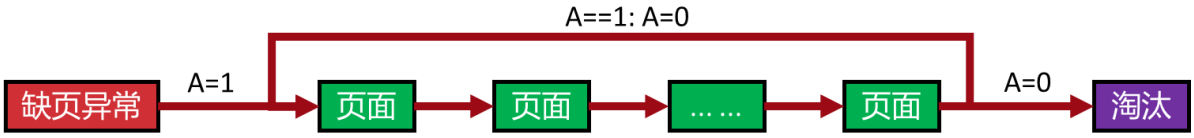
但是 FIFO本质上没有利用程序的局部性，没有栈式性质，会产生 Belady异常。它比RANDOM好得有限。

问题 我们能否利用每个页的访问位A和脏位D来优化FIFO算法?

访问位A 访问位A是可以利用的。如果我们在按FIFO规则淘汰页面时找到 一个访问位A不为0的页面，我们是把它的访问位清零后再塞到 队列开头去，直到我们找到一个访问位为0的页面 为止，这说明 它在被塞进队列里后自始至终都没被访问过，可以淘汰。

这样，就等于给FIFO算法加上了一点LRU性质。

这种带有LRU"味"但又并非严格LRU的都可以称为伪（Pesudo）LRU，即P-LRU



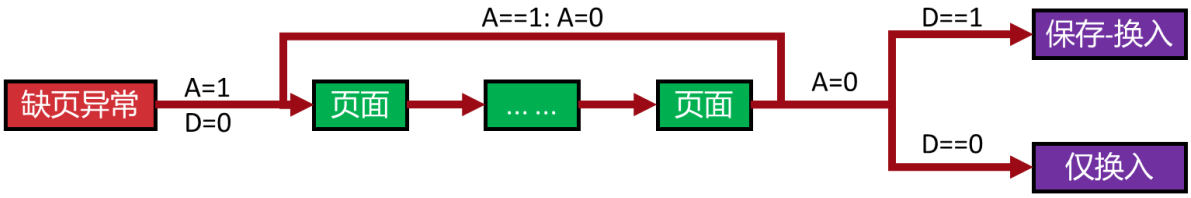
问题 一个加入队列的页如果没有被访问，经过几轮回转后被淘汰?

二次机会法 两次。第一次是A位被清零，第二次才是真正的淘汰。因此该方 法又称二次机会法。它也称CLOCK法，因为扫描是轮转进行的。

二次机会法的优化

问题 我们仅仅利用了访问位A，脏位D没利用。如何利用它?

脏位D 脏位D标志着页面是否被修改过。很多程序换入页面（比如.text 和.rodata）时，只是读取，并不修改它们，因此脏位D不置位。 这样，在把页面换出时，就不需要将它们的内容写进外存，因 为外存上那个副本仍然是和内存中一样的。这就把页面交换的 开销减小了一半。

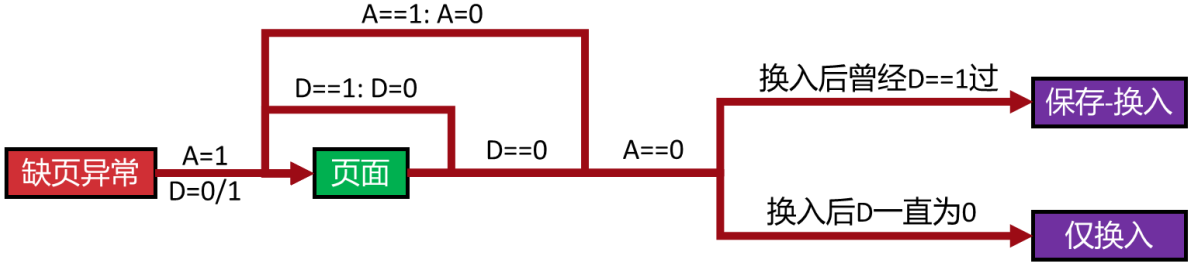


观察 D=1的页面要替换，开销是D=0的页面的一倍。况且，如果程序写了某个页面，未来很可能要读它。因此我们不是很情愿替换D==1的那些页面。

"三次机会法（非标准称呼）

解决方案 对D==1的页面，我们甚至可以再多给一次机会，将其D位清零， 并在其它地方记录下它的实际修改状态（因为我们把D位置0 了），然后再插到链表开头。

这就相当于最多给了一个页面三次机会，更贴近实际了。



问题 之前讨论都的是主存储器满时的页面替换情况，任何一个进程退出，它的所有页面都被立即释放并调配给其它进程，从而保证内存的利用率。但在现代计算机中，物理页面往往是有剩的。 这时候，如果一个进程退出，它的页面就要立刻被回收吗?

提示：考虑程序加载的动态链接库。

页面共用与缓存

页面共用 两个进程可以安全地共用一些只读页面，哪怕它们不知道这些页面被共用：比如，两个程序都加载了某个（使用位置无关代码的）动态链接库，但其.text段和.rodata段皆只需加载一份，并被两个进程的虚拟地址空间同时引用。

换言之，如果一个进程加载过某些只读页面，其它进程再加载同样的页面时就只需要将同样的物理页面添加进自己的映射， 不需要再分配新的物理内存了。

页面缓存 进程退出时，操作系统并不会立刻释放它的物理内存页（尤其是那些包含动态链接库的页面），即便它们没有被任何其它进程引用。这是考虑到长期调度的工作集：一个刚退出的程序可能在不远的未来被再次启动，而且刚引用过的动态链接库在很 靠近的未来可能会再次被引用。

这样，如果另一个用到了同样内容页面的进程被启动，它就可以直接拿着之前加载好的页面使用，而非再产生缺页异常了。

问题 上述机制减少了强制缺失还是容量缺失？你能想出几个非常有利于上述机制发挥的具体场景吗?

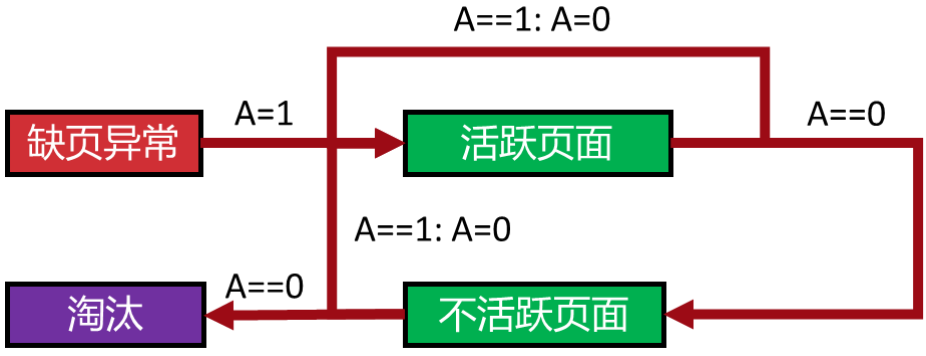
提示：软件编译、文档编辑、游戏娱乐

问题 页面共用和缓存要怎么加入到页面替换策略中？进程请求新页 面，或者分页时要调入页面，优先替换掉哪些页面?

直觉 被多个进程共用的那些页面影响面很广，一旦把它们替换出去， 对多个进程都造成影响。另外，缓存中包含的是暂时没人引用的库和数据，因此优先替换掉它们比较好。

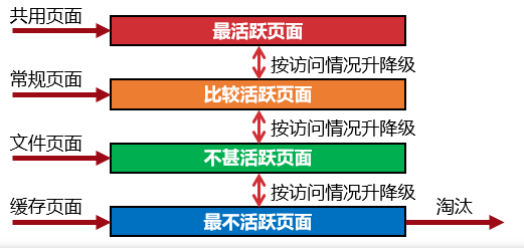
双队列PLRU 维持两个队列，一个是活跃页所在的的队列，另一个是不活跃 的页所在的队列。当要淘汰页面时，从不活跃的页所在的队列 中进行淘汰。活跃队列中的页总是慢慢地被淘汰进入不活跃的 队列，如果在不活跃的队列中它还是很久未访问，才会被淘汰。

当然，如果一个在不活跃队列中的页被访问到了，它就立即被 提升到活跃队列去。



原因 现实中的操作系统都不是简单的二次机会法，而是针对不同性 质的页采取不同的替换策略。比如，更不情愿替换那些多进程 共用的只读页，或者更倾向于替换那些没进程使用的缓存页。 这些策略都需要遍历队列，还需要考虑一个页多久没有使用； 替换某种页的倾向性越高，能容忍它不被访问的时间就越短。

使用多个队列后，不同队列中的页的活跃度不同，相当于把页 按照队列分代。对于不同的队列可采取不同的替换或升降级策 略；若定期清理和查询访问位A，甚至可具备一定的LFU性质。



页面替换的其它知识

分页文件 vs.交换分区 Windows将交换出来的页面组织成一个文件pagefile.sys放在文件 系统上，称之为“分页文件”。Linux的做法则有不同，在硬盘上创建一个连续的分区swap来存放这些页面，称之为“交换分区”。这 两种方法的目的是一致的，但组织形式有所不同。

问题 分页文件和交换分区的优劣势各有哪些?

放到外存vs.内存压缩 传统上，操作系统会将交换出来的页面放进外存。但是实际观察可以发现，部分页面的内容的重复性很高，因此可以将这些页面压缩一下，减小体积，然后仍然放在内存中。当然，此时它们无法被直接使用了，一旦被引用到，需要解压出来到真正 的物理页才能使用。

问题 将页面压缩后放进内存与把它直接放到外存有什么优势和劣势？ 你能想出结合这两方面优势的折中方案吗?

三.系统进程接口

3.1进程的基本操作

基本操作:

1. 创建 通知操作系统建立一个新的虚拟地址空间
2. 销毁 通知操作系统销毁虚拟地址空间
3. 分配资源 给进程分配更多资源

其他操作:

1. 等待 等待另一个进程被销毁
2. 复制 产生一个当前进程的副本，或者加载其他程序覆盖当前进程
3. 设置特权 赋予或撤销进程的权限
4. 设置策略 堆进程设置资源总量限制或资源分配优先级等等、

3.2进程接口实例

Linux原生接口	Windows原生接口	功能
vfork+exec函数族	CreateProcess CreateProcessAsUser	创建进程

Linux原生接口	Windows原生接口	功能
exit	ExitProcess	销毁进程
waitpid/wait4	WaitForSingleObject	等待进程
fork/clone	CreateProcess +其它系统调用	复制进程
（继承用户权限）capset ptrace mprotect	（继承用户权限） SetSecurityInfo SetProcessValidCallTargets SetProcessDEPPolicy	设置特权
setrlimit madvise	SetProcessWorkingSetSize SetProcessAffinityMask SetProcessPriorityBoost SetProcessInformation	设置策略