虚拟内存管理

背景

虚拟内存 将用户逻辑内存与物理内存分开。 这在现有物理内存有限的情况下， 为程序员提供了巨大的虚拟内存

进程的虚拟地址空间 就是进程如何在内存中存放的逻辑（ 或虚拟） 视图。

包括空白的虚拟地址空间称为稀疏 地址空间。 采用稀疏地址空间的优点是随着程序的执行， 堆栈或堆会生长或需要加载动态链接库（ 或共享对象）， 此时可以填充这些空白。

除了将逻辑内存与物理内存分开外， 虚拟内存允许文件和内存通过共享页而为多个进程所共享，好处

- 通过将共享对象映射到虚拟地址空间中， 系统库可以为多个进程所共享。 尽管每个进程都将库视为其虚拟地址空间的一部分， 但是驻留在物理内存中的库的实际页可由所有进程共享通常， 库按只读方式映射到与其链接的进程空间。
- 类似地， 虚拟内存允许进程共享内存。 如第 3 章所述， 进程之间可以通过使用共享内存来进行通信。 虚拟内存允许一个进程创建一个内存区域， 以便与其他进程共享。共享这个内存区域的进程认为： 它是其虚拟地址空间的一部分， 而事实上这部分是共享的
- 当通过系统调用 forkO 创建进程时， 可以共享页面， 从而加快进程创建。

请求调页

一种选择是， 在程序执行时将整个程序加载到物理内存。 然而， 这种方法的一个问题是， 最初可能不需要整个程序都处于内存。

另一种策略是， 仅在需要时才加载页面。 这种技术被称为请求调页

请求调页系统类似于具有交换的分页系统

这里进程驻留在外存上（ 通常为磁盘）。 当进程需要执行时， 它被交换到内存中。 不过， 不是将整个进程交换到内存中， 而是采用惰性交换器 惰 性 交 换 程序器除非需要某个页面， 否则从不将它交换到内存中。

基本概念

当换入进程时， 调页程序会猜测在该进程被再次换出之前会用到哪些页。 调页程序不是调入整个进程， 而是把那些要使用的页调入内存。 这样， 调页程序就避免了读入那些不使用的页， 也减少了交换时间和所需的物理内存空间

使用这种方案需要一定形式的硬件支持， 以区分内存的页面和磁盘的页面。

当进程执行和访问那些内存驻留 的页面时， 执行会正常进行。

如果进程试图访问那些尚未调入内存中的页面时， 情况会如何呢？ 对标记为无效的页面访问会产生缺页错误

处理这种缺页错误的程序

1.检查这个进程的内部表（ 通常与- PCB ）一起保存），以确定该引用是有效的还是无效的内存访问。
2.如果引用无效， 那么终止进程。 如果引用有效但是尚未调入页面， 那么现在就应调入。
3.找到一个空闲帧（ 例如， 从空闲帧链表上得到一个）
4.调度一个磁盘操作， 以将所需页面读到刚分配的帧。
5.当磁盘读取完成时， 修改进程的内部表和页表， 以指示该页现在处于内存中。
6.重新启动被陷阱中断的指令。 该进程现在能访问所需的页面， 就好像它总是在内存中。

支持请求调页的硬件与分页和交换的硬件相同：

- 页表
- 外存

请求调页的关键要求是在缺页错误后重新启动任何指令的能力。 因为当发生缺页错误时， 保存了被中断的进程状态（ 寄存器、 条件代码、 指令计数器）， 所以应能够在完全相同的位置和状态下， 重新启动进程，

当一条指令可以修改多个不同的位置时， 就会出现重要困难

一种解决方案是， 微代码计算并试图访问两块的两端。 如果会出现缺页错误， 那么在这一步就会出现（ 在任何内容被修改之前 ）。 然后可以执行移动。 我们知道不会发生缺页错误， 因为所有相关页面都在内存中。

另一个解决方案使用临时寄存器来保存覆盖位置的值。 如果有缺页错误， 则在陷阱发生之前， 所有旧值都将写回到内存中。 该动作将内存恢复到指令启动之前的状态， 这样就能够重复该指令

请求调页的性能

有效访问时间，设p为发生缺页中断的概率希 望 p 接近于 0, 即缺页错误很少。 那么有效访问时间为 有效访问时间 =(1-p)x ma + p x 缺页错误时间

写时复制

它通过允许父进程和子进程最初共享相同的页面来工作。 这些共享页面标记为写时复制这意味着，如果任何一个进程写入共享页面， 那么就创建共享页面的副本

当确定采用写时复制来复制页面时， 重要的是注意空闲页面的分配位置。 许多操作系统为这类请求提供了一个空闲的页面池当进程的堆栈或堆要扩展时或有写时复制页面需要管理时， 通常分配这些空闲页面。 操作系统分配这些页面通常采用称为按需填零

页面置换

基本页面置换

1.找到所需页面的磁盘位置。
2.找到一个空闲帧：
a.如果有空闲帧， 那么就使用它。
b.如果没有空闲帧， 那么就使用页面置换算法来选择一个牺牲帧
c.将牺牲帧的内容写到磁盘上， 修改对应的页表和帧表。
3.将所需页面读入（新的） 空闲帧， 修改页表和帧表。
4.从发生缺页错误位置， 继续用户进程。

采用修改位 (或脏位) 可减少这种开销。 当采用这种方案时， 每个页面或帧都有一个修改位， 两者之间的关联采用硬件。 每当页面内的任何字节被写入时，它的页面修改位会由硬件来设置， 以表示该页面已被修改过。

为实现请求调页， 必须解决两个主要问题： 应设计帧分配算法和页面置换算法

可以这样评估一个算法： 针对特定内存引用串， 运行某个置换算法， 并计算缺页错误的数量。

FIFO 页面置换

FIFO 页面置换算法为每个页面记录了调到内存的时间。 当必须置换页面时， 将选择最旧的页面。

帧的缺页错误数 10) 比 3 帧的缺页错误数 9) 还要大！ 这个最意想不到的结果被称为 Belady 异常，对于有些页面置换算法， 随着分配帧数量的增加，缺页错误率可能也会增加

最优页面置换

这个算法具有所有算法的最低的缺页错误率， 并且不会遭受 Belady 异常。 这种算法确实存在它被称为 OPT 或 MIN 简单地说：置换最长时间不会使用的页面，这种页面置换算法确保对于给定数量的帧会产生最低的可能的缺页错误率。

LRU 页面置换

LRU 置换将每个页面与它的上次使用的时间关联起来。 当需要置换页面时， LRU 选择最长时间没有使用的页面。

近似 LRU 页面置换

额外引用位算法，第二次机会算法，增强型第二次机会算法

基于计数的页面置换

最不经常使用 LFU) 页面置换算法要求置换具有最小计数的页面。

最经常使用MFU) 页面置换算法是基于如下论点： 具有最小计数的页面可能刚刚被引入并且尚未使用

页面缓冲算法

系统通常保留一个空闲帧缓冲池。 当出现缺页错误时， 会像以前一样选择一个牺牲帧。 然而， 在写出牺牲帧之前， 所 需页面就读到来自缓冲池的空闲帧。

这种方法的扩展之一是， 维护一个修改页面的列表。 每当调页设备空闲时， 就选择一个修改页面以写到磁盘上，然后重置它的修改位

另一种修改是， 保留一个空闲帧池， 并且记住哪些页面在哪些帧内。 因为在帧被写到磁盘后帧内容并未被修改， 所以当该帧被重用之前， 如果再次需要， 那么旧的页面可以从空闲帧池中直接取出并被使用。

应用程序与页面置换

有的操作系统允许特殊程序能够将磁盘分区作为逻辑块的大的数组来使用， 而不需要通过文件系统的数据结构。 这种数组有时称为原始磁盘

帧分配

帧的最小数

帧分配策略受到多方面的限制。 例如， 所分配的帧不能超过可用帧的数量（ 除非有页面共享）， 也必须分配至少最小数量的帧。

分配至少最小数量的帧的一个原因涉及性能。 显然， 随着分配给每个进程的帧数量的减少， 缺页错误率增加， 从而减慢进程执行。 此外， 请记住， 若在执行指令完成之前发生缺页错误， 应重新启动指令。 因此， 必须有足够的帧来容纳任何单个指令可以引用的所有不同的页面。

分配算法

平均分配，比例分配

全局分配与局部分配

为各个进程分配帧的另一个重要因素是页面置换。 由于多个进程竞争帧， 可以将页面置换算法分为两大类： 全局置换 和局部置换

全局置换允许一个进程从所有帧的集合中选择一个置换帧， 而不管该帧是否已分配给其他进程；也就是说， 一个进程可以从另一个进程那里获取帧。

局部置换要求每个进程只从它自己分配的帧中进行选择。

非均匀内存访问

具有明显不同的内存访问时间的系统统称为非均匀内存访问（ NUMA ）系统，并且毫无例外地， 它们要慢于内存和 CPU 位于同一主板的系统

管理哪些页面帧位于哪些位置能够明显影响 NUMA 系统的性能。

算法修改包括让调度程序跟踪每个进程运行的最后一个 CPU。

系统抖动

没有 “足够” 帧的进程。 如果进程没有需要支持活动使用页面的帧数， 那么它会很快产生缺页错误。 此时， 必须置换某个页面。 然而， 由于它的所有页面都在使用中， 所以必须立即置换需要再次使用的页面。 因此， 它会再次快速产生缺页错误， 再一次置换必须立即返回的页面， 如此快速进行。

这种高度的页面调度活动称为抖动（ thrashing）。 如果一个进程的调页时间多于它的执行时间， 那么这个进程就在抖动。

系统抖动的原因

CPU 调度程序看到 CPU 利用率的降低， 进而会增加多道程度。 新进程试图从其他运行进程中获取帧来启动， 从而导致更多的缺页错误和更长的调页设备队列。 因此， CPU 利用率进一步下降， 并且 CPU 调度程序试图再次增加多道程度。 这样就出现了抖动，

工作集模型

工作集模型 是基于局部性假设的。 这个模型采用参数A 定义工作集窗口。 它的思想是检查最近 A 个页面引用。 这最近 A个页面引用的页面集合称为工作集 如果一个页面处于活动使用状态， 那么它处在工作集中。 如果它不再使用， 那么它在最后一次引用的 A 时间单位后， 会从工作集中删除。

缺页错误频率

抖动具有高缺页错误率。 因此， 需要控制缺页错误率。 当缺页错误率太高时，我们知道该进程需要更多的帧。相反， 如果缺页错误率太低， 则该进程可能具有太多的帧。我们可以设置所需缺页错误率的上下限

内存映射文件

基本机制

实现文件的内存映射是， 将每个磁盘块映射到一个或多个内存页面。 最初， 文件访问按普通请求调页来进行， 从而产生缺页错误。 这样， 文件的页面大小部分从文件系统读取到物理页面

内存映射 I/O

专用 I/O 指令允许在这些寄存器和系统内存之间进行数据传输。 为了更方便地访问 I/O 设备， 许多计算机体系结构提供了内存映射 I/O 在这种情况下，一组内存地址专门映射到设备寄存器。 对这些内存地址的读取和写入， 导致数据传到或取自设备寄存器。

分配内核内存

用于分配内核内存的空闲内存池通常不同于用于普通用户模式进程的列表。 这有两个主要原因

- 内核需要为不同大小的数据结构请求内存， 其中有的小于一页。 因此， 内核应保守地使用内存， 并努力最小化碎片浪费。 这一点非常重要， 因为许多操作系统的内核代码或数据不受调页系统的控制。
- 用户模式进程分配的页面不必位于连续物理内存。 然而， 有的硬件设备与物理内存直接交互， 即无法享有虚拟内存接口带来的便利， 因而可能要求内存常驻在连续物理内存中。

伙伴系统

伙伴系统 从物理连续的大小固定的段上进行分配。 从这个段上分配内存， 采用 2 的幂分配器来满足请求分配单元的大小为 2 的幂（ 4KB 、8KB、 16KB 等）。 请求单元的大小如不适当， 就圆整到下一个更大的 2 的幂。

伙伴系统的一个优点是： 通过称为合并 的技术， 可以将相邻伙伴快速组合以形成更大分段。

伙伴系统的明显缺点是： 由于圆整到下一个 2 的幂， 很可能造成分配段内的碎片。

slab 分配

每个 slab 由一个或多个物理连续的页面组成。 每个 cache 由一个或多个 slab 组成。 每个内核数据结构都有一个cache,

在 Linux 中， slab 可以处于三种可能状态之一：

- 满的（ fiill): slab 的所有对象标记为使用。
- 空的（ empty): slab 上的所有对象标记为空闲。
- 部分（ partial): slab 上的对象有的标记为使用， 有的标记为空闲.

slab 分配器提供两个主要优点：

没有因碎片而引起内存浪费。 碎片不是问题， 因为每个内核数据结构都有关联的 cache, 每个 cache 都由一个或多个 slab 组成， 而 slab 按所表示对象的大小来分块。

可以快速满足内存请求。 因此， 当对象频繁地被分配和释放时， 如来自内核请求的情况， slab 分配方案在管理内存时特别有效。