虚拟 内存

第二十一章——超越 物理 内存

刘 玉峰

Fx_163.com 湖南

大学

- 到 目前为止， 我们一直 假设 地址 空间小得不切实际，只能容纳在物理 内存中。

- 然而， 为了 支持 大的地址 空间， 操作系统 将**需要一个地方来存放目前需求不大的 部分地址空间**。

- 通常 ， 这样的位置应该具有**比存储器更大的容量**。在 现代系统中，这个 角色通常由**硬盘 驱动器提供**。

交换 空间

- 我们需要做的 第一 件事　　是 在磁盘上**保留 一些空间**　，以便来回移动 页面　。

- 在 操作 系统中，　我们 通常　将 这样的空间 称为 交换 空间 。

- 因此，　我们 将 简单地假设　OS 可以以页面大小的单位从交换空间读取　和写入交换空间。

- 操作 系统　需要　记住给定页面的 **磁盘 地址**　。

只有 少量 物理内存 可用时的大型 虚拟机

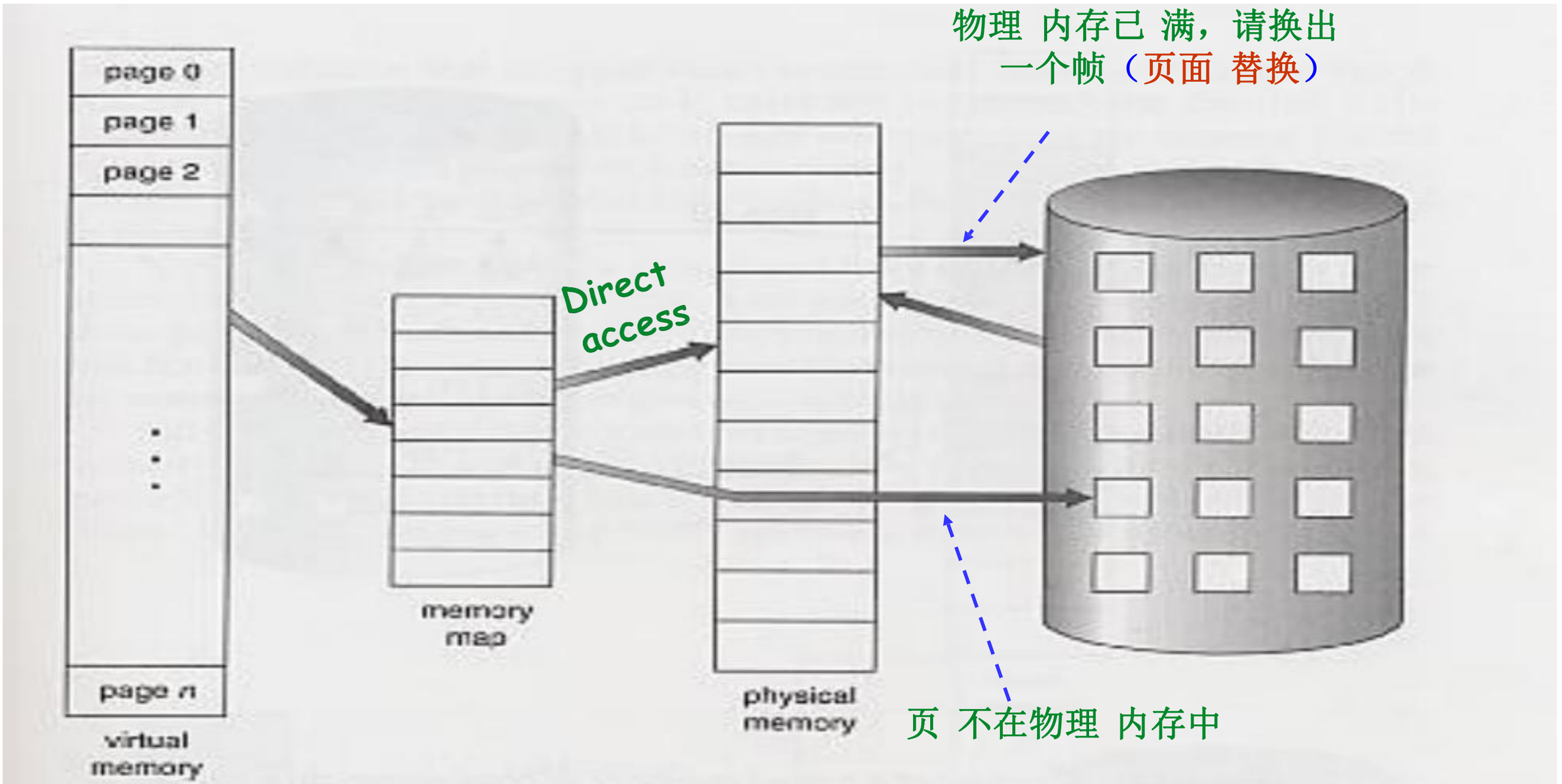
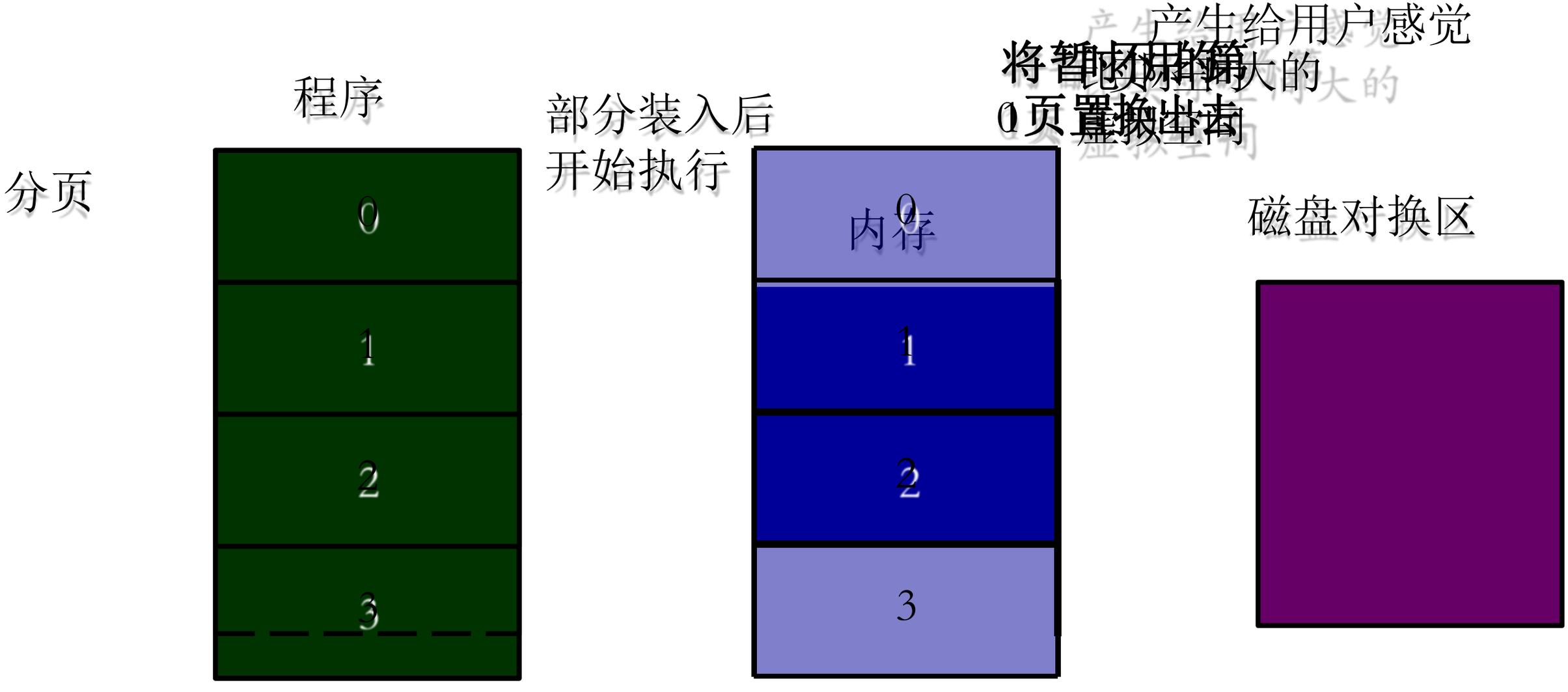


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

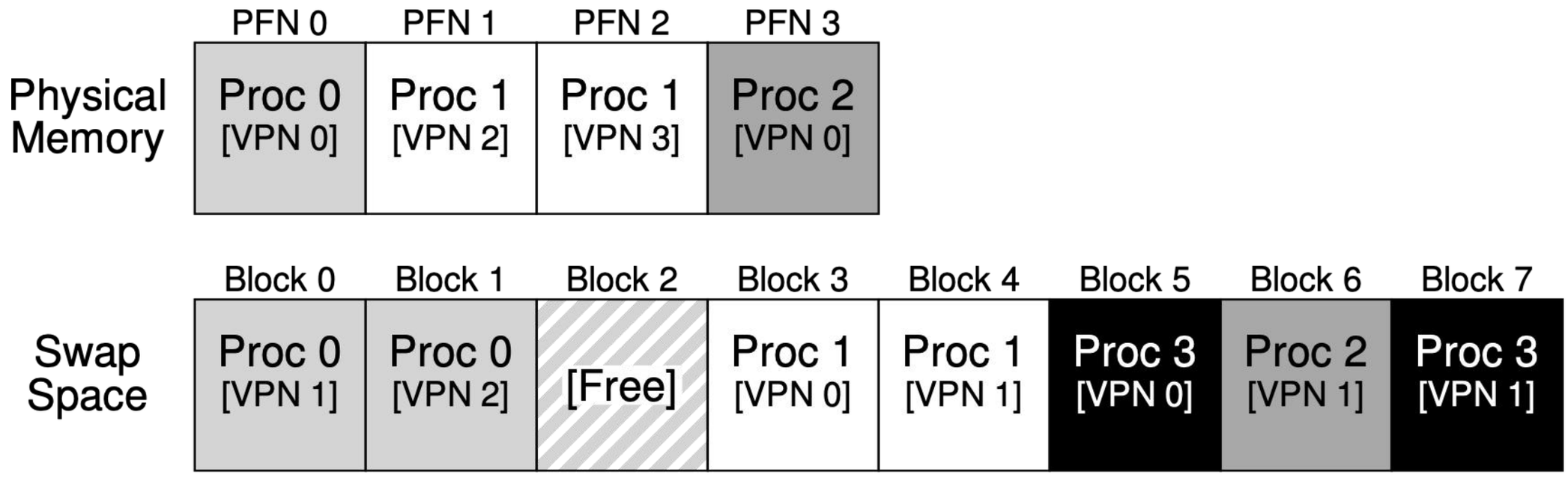
虚拟存储器的定义

- 虚拟存储器的定义
  - 虚拟存储指仅把作业的一部分装入内存便可运行的存储管理系统，通过作业各部分的动态调入和置换，用户所感觉的存储空间比实际空间大，称之为虚空间。



虚空间大小

- 虚空间大小
  - 虚空间的逻辑大小 = 可寻址范围
  - 虚空间的实际大小 = 内存+外存对换区
- 例：32位操作系统的可寻址范围是 2^{32}=4GByte，Windows98系列系统。
- 例：在window系统盘根目录下，有兑换文件--外存对换区。如XP系统的 pagefile.sys 文件
- 例：Linux中的swap交换分区



- 本例中，一个 包含4 页的物理内存和一个 8 页的交换空间。
- 3 个进程（进程 0、进程 1 和进程 2）主动共享物理内存。但 3 个中的每一个，都只有一部分有效页在内存中剩下的在硬盘的交换空间中。
- 第 4 个进程（进程 3）的所有页都被交换到硬盘上。
- 有一块交换空间是空闲的。

存在位

- 在系统的更高位置添加一些机器，以支持在磁盘上交换页面。
  - 当硬件在PTE中查找时，它可能会发现该页不存在于物理内存中。

- 访问不在物理存储器中的页面的动作通常被称为**页面错误**。

价值	含义
1	页存在于物理内存中
0	页面不在内存中，而是在磁盘上。

页面 错误

- 如果 页面 不 存在 并且 已经 交换 到 磁盘， 则 操作 系统 将需要 将 页面交换到 内存中。

- 因此，出现了 一个问题： **操作系统如何知道在哪里 找到所需的页面**？

- 在 许多 系统中，页表是存储此类信息的自然位置。因此，OS可以使用PTE中通常用于数据的位， 诸如**用于盘地址的页的PFN** 。

- 当 OS 接收到 一 个 页面的页面错误时， 它 会 在 PTE 中查找地址，并向磁盘发出请求，将页面提取到内存中 。

存在位与外存地址

- 存在位
- 对页表进行扩充
  - -让系统了解页面装入状态

页面如果不在内存中，记录页面在交换区的位置

页号	块号	存取控制	存在位	引用位	修改位	外存地址

页面是否被访问过

页面是否被修改过

-存在位：

»为0　不在内存中

»为1　在内存中

补充：交换术语及其他

对于不同的机器和操作系统，虚拟内存系统的术语可能会有点令人困惑和不同。例如，页错误（page fault）一般是指对页表引用时产生某种错误：这可能包括在这里讨论的错误类型，即页不存在的错误，但有时指的是内存非法访问。事实上，我们将这种完全合法的访问（页被映射到进程的虚拟地址空间，但此时不在物理内存中）称为“错误”是很奇怪的。实际上，它应该被称为“页未命中（page miss）”。但是通常，当人们说一个程序“页错误”时，意味着它正在访问的虚拟地址空间的一部分，被操作系统交换到了硬盘上。

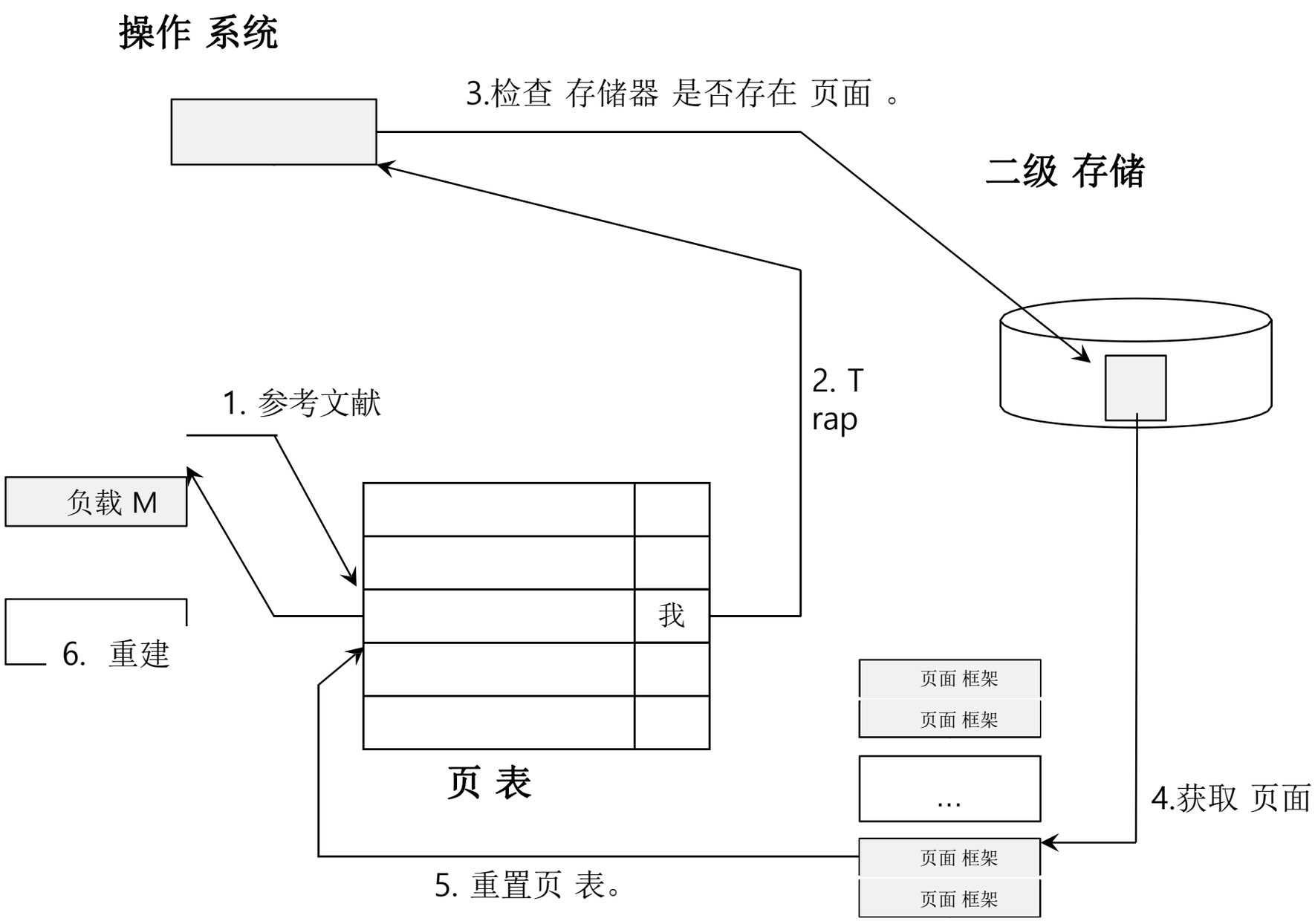
我们怀疑这种行为之所以被称为“错误”，是因为操作系统中的处理机制。当一些不寻常的事情发生的时候，即硬件不知道如何处理的时候，硬件只是简单地把控制权交给操作系统，希望操作系统能够解决。在这种情况下，进程想要访问的页不在内存中。硬件唯一能做的就是触发异常，操作系统从开始接管。由于这与进程执行非法操作处理流程一样，所以我们把这个活动称为“错误”，这也许并不奇怪。

页面 错误

- 访问 不在物理内存中的页。
- 如果 页面不存在并且已经被交换了磁盘， 则 OS 需要 将 页面交换到 存储器中以便服务页面 错误。

页面 故障控制 流程

- PTE用于数据，如磁盘地址页 的 PFN 。



当操作系统 接收 到 页面错误时，它会在PTE中查找并向磁盘发出请求。

页面 故障控制流程-硬件

```
1:     VPN = (VirtualAddress&VPN_MASK) 〉〉 SHIFT
二:     (Success, TlbEntry) =TLB_Lookup (VPN)
三:    if  (Success == True)  // TLB命中
四:    if  (CanAccess (TlbEntry.ProtectBits) == True)
五:           Offset = VirtualAddress&OFFSET_MASK
六:           PhysAddr = (TlbEntry.PFN<<SHIFT) |偏移寄存器
七:           =AccessMemory (PhysAddr)
八:    else RaiseException ( PROTECTION_FAULT)
```

页面 故障控制流程-硬件件

```
九:     else// TLB 未命中
十:     PTEAddr = PTBR +（VPN*sizeof（PTE））
11:     PTE = AccessMemory（PTEAddr）
十二:   if （PTE.Valid == False）   //还记得PTE的有效位吗？虚拟内存中的无效页，无需分配物理内存
十三:           RaiseException（SEGMENTATION_FAULT）
十四:   否则
十五:   if（CanAccess（PTE.ProtectBits）==False）
16:             RaiseException（ PROTECTION_FAULT）
十七:   else if （PTE.Present == True） //数据页在内存
18:     //假设硬件管理的 TLB
十九:           TLB_Insert（VPN，PTE.PFN，PTE.ProtectBits）
20:             RetryInstruction（）
二十一:         else if （PTE.Present == False） // //数据页不在内存
二十三:         RaiseException（ PAGE_FAULT）
```

页面 故障控制流程- 软件

```
1:      PFN =FindFreePhysicalPage（）
二:     if （PFN == -1） //没有找到空闲页
三:          PFN = EvictPage（） //运行替换算法
四:          DiskRead（PTE.DiskAddr，pfn）// sleep（等待I/O）
五:          PTE.present =True //使用当前PTE更新页表。PFN = PFN//位和转
六:          换（PFN）RetryInstruction（） //重试指令
七:
```

- 首先， 操作系统必须为将要换入的页找到一个物理帧，如果没有这样的物理帧，我们将不得不等 待交换算法运行，并从内存中踢出一些页，释放帧供这里使用。在获得物理帧后，处理程 序发出 I/O 请求从交换空间读取页。最后，当这个慢操作完成时，操作系统更新页表并重试 指令。重试将导致 TLB 未命中，然后再一次重试时，TLB 命中，此时硬件将能够访问所需 的值

如果内存已满怎么办?

- 操作 系统 喜欢 调出 页面 来 为 操作系统 即将 引入 的新页面腾出空间。
- 选择 要 踢出或替换的页面的过程称为页面替换策略

当 更换真正发生时

- 操作系统不会等到内存已经完全满了以后才执行交换流程，操作系统可以更主动地预留一小部分空闲内存。
- 为了保证有少量的空闲内存，大多数操作系统会设置高水位线（High Watermark，HW） 和低水位线（Low Watermark，LW），来帮助决定何时从内存中清除页。
- 当操 作系统发现有少于 LW 个页可用时，后台负责释放内存的线程会开始运行，直到有 HW 个 可用的物理页。这个后台线程有时称为交换守护进程（swap daemon）或页守护进程（page daemon）
- 通过同时执行多个交换过程，我们可以进行一些性能优化。例如，许多系统会把多个 要写入的页聚集（cluster）或分组（group），同时写入到交换区间，从而提高硬盘的效率[LL82]。 我们稍后在讨论硬盘时将会看到，这种合并操作减少了硬盘的寻道和旋转开销，从而显著 提高了性能。

如果内存已满怎么办?

- 我们 **假设有足够的空闲 内存** 可 供 从 交换 空间分页。

- 内存 **可能已满**（或接近满）。因此，OS可能 想要首先 调出 一个或 多个页 以为新 页腾出空间。

- 选择 要 踢 出 或 替换的页面的过程称为**页面替换 策略**。

关键问题：如何决定踢出哪个页

操作系统如何决定从内存中踢出哪一页（或哪几页）？这个决定由系统的替换策略做出，替换策略通常会遵循一些通用的原则（下面将会讨论），但也会包括一些调整，以避免特殊情况下的行为。

第二十二章——缓存管理

- 主存储器可以被视为系统中虚拟存储器页面的高速缓存。

- 此缓存的替换策略是**最小化缓存未命中的数量**。

- 或者，可以将我们的目标视为**最大化缓存命中**的数量。

- Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT** 平均内存访问时间) for a program.

$$AMAT = (Hit\% \cdot T_M) + (Miss\% \cdot T_D)$$

TM表示访问内存的成本
TD表示访问磁盘的成本
Hit表示在缓存中找到数据的 概率（命中）
Miss表示在缓存中找不到数据的概率（未命中）。

- 假设 访问内存的开销（T_M） 约为**100纳秒**， 访问磁盘的 开销 （T_D）约为 **10毫秒**， 未命中率为**10%**，命中率为**90%**。

- 我们 有 以下 AMAT： 0.9 x 100ns + 0.1 x 10ms， 即 90ns + 1ms， 或 1.00009 ms， 或 约 1 毫秒。

- 如果 我们的命中率是**99.9%**，结果就完全不同了：AMAT是10.1μs，**大约快100倍**。当命中率接近100%时，AMAT接近100纳秒。

最 优 替换策略

- 导致 最少的失误
  - 替换　将来 访问距离最远 的页
  - 导致尽可能少的缓存 未命中
- Serve only as a comparison point, to know how close we are to perfect（由于这是面向未来的算法，因此不可能真正实现，只能作为一个参照标准）

在引用最远将来会访问的页之前，你肯定会引用其他页!

追踪 最优 策略

参考 行

0 1 2 0 1 3 0 3 1 2 1

前 3 个访问是未命中，因为缓存开始是空的。这种未命中有时也称作冷启动未命中（cold-start miss，或强制未命中，compulsory miss）。

访问	命中/未命中？	驱逐	产生的 缓存 状态
0	小姐		0
1	小姐		0,1
2	小姐		0,1,2
0	命中		0,1,2
1	命中		0,1,2
3	小姐	2	0,1,3
0	命中		0,1,3
3	命中		0,1,3
1	命中		0,1,3
2	小姐	3	0,1,2
1	命中		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

未来 是 未知 的。

- 不幸的是， **未来并不是普遍 知道的**;您不能为通用操作 系统构建最优策略。

- 因此， 最优策略只能作为**一个比较**点， 以 了解**我们 离“完美”有多近。**

提示：与最优策略对比非常有用

虽然最优策略非常不切实际，但作为仿真或其他研究的比较者还是非常有用的。比如，单说你喜欢的新算法有80%的命中率是没有意义的，但加上最优算法只有82%的命中率（因此你的新方法非常接近最优），就会使得结果很有意义，并给出了它的上下文。因此，在你进行的任何研究中，知道最优策略可以方便进行对比，知道你的策略有多大的改进空间，也用于决定当策略已经非常接近最优策略时，停止做无谓的优化[AD03]。

追踪 FIFIO 策略

参考 行

0 1 2 0 1 3 0 3 1 2 1

访问	命中/未命中？	驱逐	产生的 缓存 状态
0	小姐		0
1	小姐		0,1
2	小姐		0,1,2
0	命中		0,1,2
1	命中		0,1,2
3	小姐	0	1,2,3
0	小姐	1	2,3,0
3	命中		2,3,0
1	小姐		3,0,1
2	小姐	3	0,1,2
1	命中		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

即使 页面0已经被访问了 许多
但 FIFO仍然会把它踢出局。

补充：Belady 的异常

Belady（最优策略发明者）及其同事发现了一个有意思的引用序列[BNS69]。内存引用顺序是：1，2，3，4，1，2，5，1，2，3，4，5。他们正在研究的替换策略是 FIFO。有趣的问题：当缓存大小从 3 变成 4 时，缓存命中率如何变化？

一般来说，当缓存变大时，缓存命中率是会提高的（变好）。但在这个例子，采用 FIFO，命中率反而下降了！你可以自己计算一下缓存命中和未命中次数。这种奇怪的现象被称为 Belady 的异常（Belady's Anomaly）。

其他一些策略，比如 LRU，不会遇到这个问题。可以猜猜为什么？事实证明，LRU 具有所谓的栈特性（stack property）[M+70]。对于具有这个性质的算法，大小为 $N+1$ 的缓存自然包括大小为 N 的缓存的内容。因此，当增加缓存大小时，缓存命中率至少保证不变，有可能提高。先进先出（FIFO）和随机（Random）等显然没有栈特性，因此容易出现异常行为。

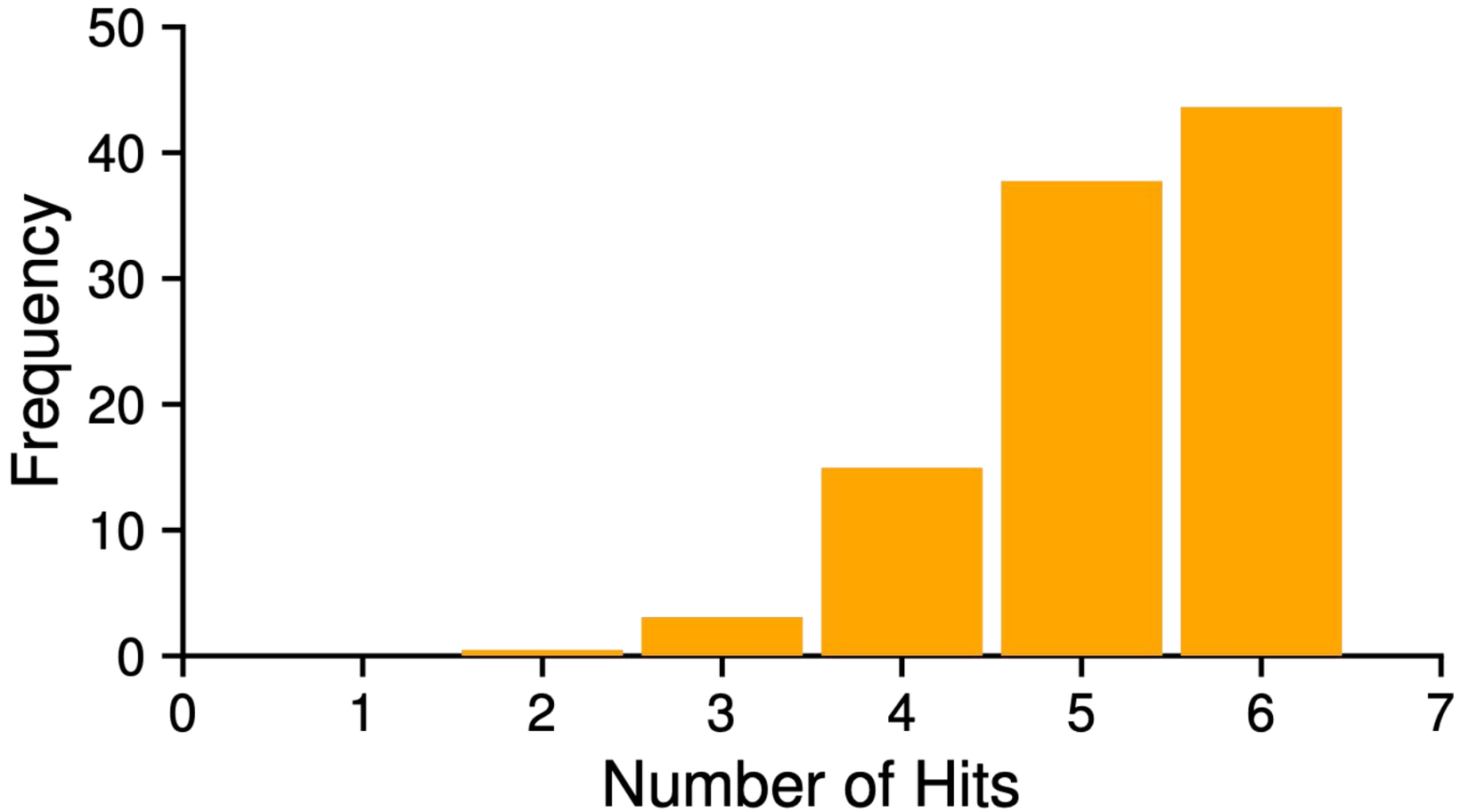
另一个 简单的策略：随机

- Random 具有类似于FIFO的属性;它实现起来很**简单**，但在选择要驱逐的块时并没有真正尝试过智能。

- 随机的 表现完全取决 于随机在选择中的 幸运（或不幸）程度。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Tracing the Random Policy



Random Performance over 10,000 Trials

- 当然，随机的表现完全取决于多幸运（或不幸）。在上面的例子中，随机比 FIFO 好一点， 比最优的差一点。事实上，我们可以运行数千 次的随机实验，求得一个平均的结果。
- 图 22.1 显示了 10000 次试验后随机策略的平均命中 率，每次试验都有不同的随机种子。正如你所 看到的，有些时候（仅仅 40%的概率），随机 和最优策略一样好，在上述例子中，命中内存的次数是 6 次。有时候情况会更糟糕，只有 2 次或更少。随机策略取决于当时的运气。

使用 历史：LRU

- 不幸的是， 任何　像 FIFO 或 Random这样简单的策略都可能有一个共同的问题：它**可能会 踢 出** 一　个即将 被再次引用的重要页面。

- 正如 我们 对 **调度 策略所做的那样**， 为了 改善 我们对未来的猜测，我们再次 依靠 过去并使用历史作为我们的指南。

- 例如， 如果 一个 程序在不久的过去访问过 一个 页面　，那么 它 很 可能　在 不久的 将来再次访问它。

- 页替换策略可以使用的一个历史信息是频率（frequency）。如果一个页被访问了很多次， 也许它不应该被替换，因为它显然更有价值。
- 页更常用的属性是访问的近期性（recency）， 越近被访问过的页，也许再次访问的可能性也就越大。
- 这一系列的策略是基于人们所说的局部性原则（principle of locality）[D70]，基本上只是对程序及其行为的观察。这个原理简单地说就是程序倾向于频繁地访问某些代码（例如 循环）和数据结构（例如循环访问的数组）。因此，我们应该尝试用历史数据来确定哪些页 面更重要，并在需要踢出页时将这些页保存在内存中
- 一系列简单的基于历史的算法诞生了。“最不经常使用”（Least-Frequently-Used， LFU）策略会替换最不经常使用的页。同样，“最少最近使用”（Least-Recently-Used，LRU） 策略替换最近最少使用的页面。.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Tracing the LRU Policy

- 给某作业分配了三块主存（开始时为空），采用先进先出页面置换算法，该作业依次访问的页号为：1，2，3，4，1，2，5，1，2，3，4，5，6,将产生（　）次缺页中断。
- 如果采用LRU，将会产生（　）次缺页中断。
- 如果是最优置换，将会产生（　）次缺页中断。

实现 历史算法

- 为了 跟踪哪些页面最近和最少被 使用， 系统 必须对每个内存引用进行 一些 统计 工作 。

- 有一 种方法 可以 帮助 加快 这 一速度，那就是 添加一 点硬件支持，比如在内存中添加一个**时间字段**。

- 当 替换 一个 页面时， 操作系统 只需 **扫描系统中的所有 时间字段**，以找到最近最少使用的页面。

- 不幸的是， 随着 系统中页面数量 的增长，为了找到最近使用最少的页面而扫描大量时间的代价**非常 昂贵**。

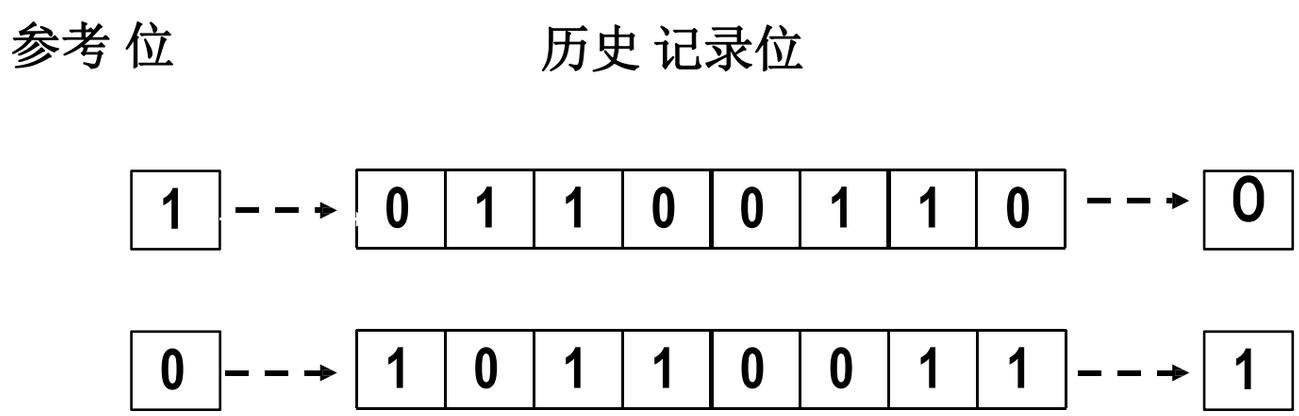
关键问题：如何实现 LRU 替换策略

由于实现完美的 LRU 代价非常昂贵，我们能否实现一个近似的 LRU 算法，并且依然能够获得预期的效果？

随着系统中页数量的增长，扫描所有页的时间字段只是为了找到最精确最少使用的页，这个代价太昂贵。想象一下一台拥有 4GB 内存的机器，内存切成 4KB 的页。 这台机器有一百万页，即使以现代 CPU 速度找到 LRU 页也将需要很长时间。这就引出了一 个问题：我们是否真的需要找到绝对最旧的页来替换？找到差不多最旧的页可以吗?

LRU近似算法

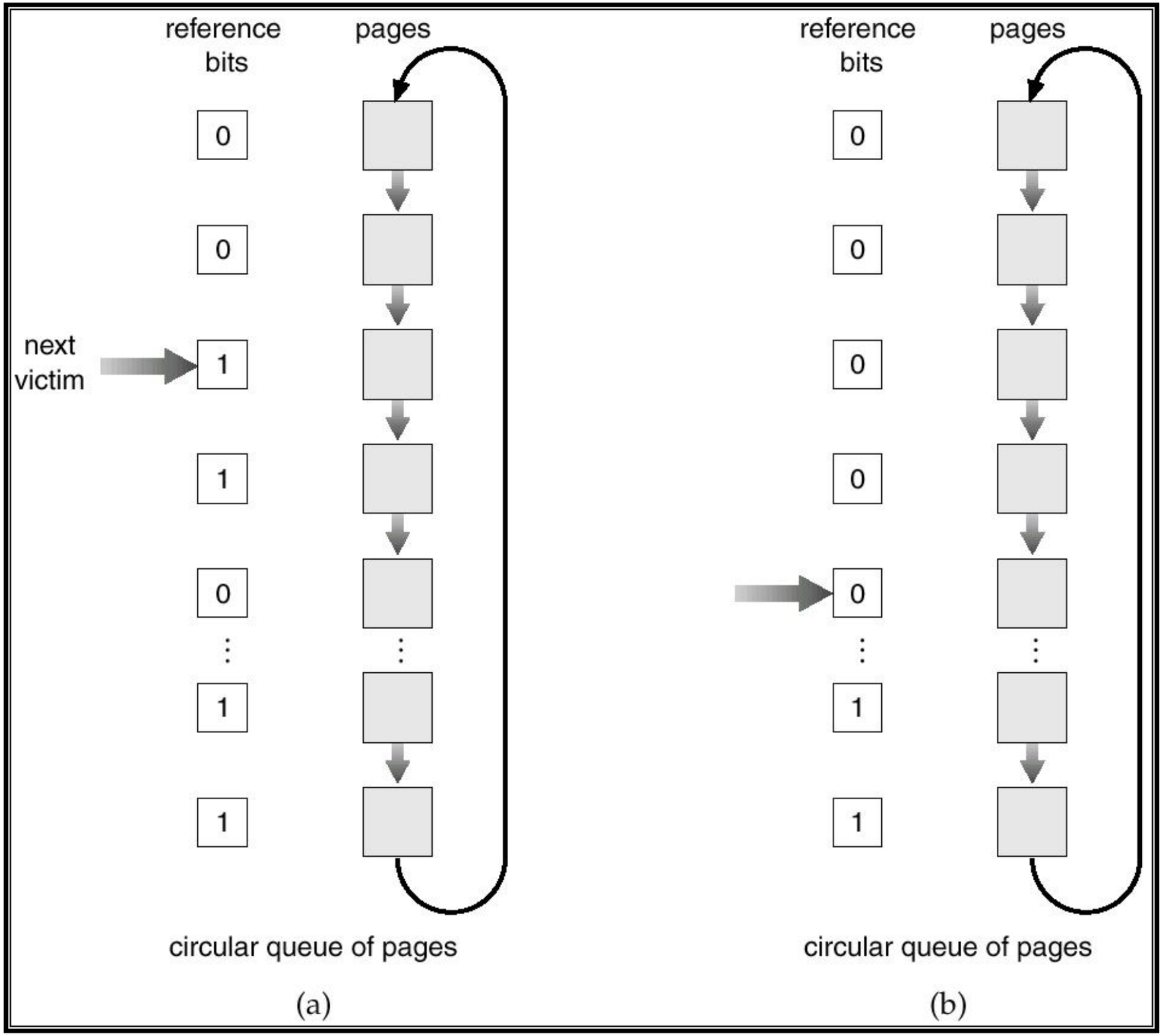
- 参考 位
  - 与 每个 页面 关联 一 个位， 初始 -= 0
  - When page is referenced bit set to 1 at （在每个时间间隔范围内，当页被访问时设置为1）.
- 编号最小的 页面 是 LRU 页面;



时钟 算法

- 系统中的所有页都放在一个循环列表中。时 钟指针（clock hand）开始时指向某个特定的页（哪个页不重要）。
- 当必须进行页替换时，操 作系统检查当前指向的页 P 的使用位是 1 还是 0。如果是 1，则意味着页面 P 最近被使用， 因此不适合被替换。然后，P 的使用位设置为 0，时钟指针递增到下一页（P + 1）。
- 该算法 一直持续到找到一个使用位为 0 的页，使用位为 0 意味着这个页最近没有被使用过（在最 坏的情况下，所有的页都已经被使用了，那么就将所有页的使用位都设置为 0）。

Second-Chance （clock）页面替换算法



改进 的二次机会算法

由访问位A和修改位M可以组合成下面四种类型的页面：

1类(A=0, M=0): 表示该页最近既未被访问， 又未被修改， 是最佳淘汰页。

2类(A=0, M=1)： 表示该页最近未被访问， 但已被修改， 并不是很好的淘汰页。

3类(A=1, M=0)： 最近已被访问， 但未被修改， 该页有可能再被访问。

4类(A=1, M=1): 最近已被访问且被修改， 该页可能再被访问。

如果页已被修改（modified）并因此变脏（dirty），则踢出它就必须将它写回磁盘，这很昂贵。如果它没有被修改（因此是干净的，clean），踢出就没成 本。物理帧可以简单地重用于其他目的而无须额外的 I/O。因此，一些虚拟机系统更倾向于踢出干净页，而不是脏页

其他 VM 策略

- 除了按需请求调页。
- 操作系统可能会猜测一个页面即将被使用，从而提前载入。这种行为被称为预取（prefetching），只有在有合理的成功机会时才应该这样做。例如，一些系统将假设如果代码页 P 被载入内存，那么代码页 P + 1 很可能很 快被访问，因此也应该被载入内存.

抖动

- 当内存就是被超额请求时，操作系统应该做 什么，这组正在运行的进程的内存需求是否超出了可用物理内存？在这种情况下，系统将不断地进行换页，这种情况有时被称为抖动.
- 一些早期的操作系统有一组相当复杂的机制，以便在抖动发生时检测并由系统决定不运行部分进程，从而减少进程的工作集.
- 某些版 本的 Linux 会运行“内存不足的杀手程序（out-of-memory killer）”。这个守护进程选择一个 内存密集型进程并杀死它，从而以不怎么委婉的方式减少内存。.

磁盘存储器地址

访磁盘存储器

命中

外部地址变换
虚页号→磁盘实地址

未命中

外部地址变换

U+P

U | P | D

Av 多用户虚地址

主存页面失效

U+P

未命中

内部地址变换
虚页号→主存实页号

命中

访问主存

p | d

A

主存页面表

主存未满

主存满

页面
替换算法

主存页号

I/O处理机
（I/O通道）

调入页

被替换页

调入页

替换页

0页
1页
…
…
2^p-1

主存储器

0页
1页
…
…
2^P-1

磁盘存储器

逻页

0页
1页
…
…
2^P-1
0页
1页

…
…

X用户

Y用户

结束